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

Under the supervision of Graeme Smith

Submitted on XXXX

Table of Contents

[1 Introduction 1](#_Toc85019292)

[1.1 Topic 1](#_Toc85019293)

[1.2 Purpose 1](#_Toc85019294)

[1.3 Goals 1](#_Toc85019295)

[1.4 Scope 1](#_Toc85019296)

[2 Background 2](#_Toc85019297)

[2.1 Formal Methods 2](#_Toc85019298)

[2.2 Weakest Precondition Proofs and Hoare Logic 2](#_Toc85019299)

[2.2.1 Hoare Triples 3](#_Toc85019300)

[2.2.2 Assignment Axiom 3](#_Toc85019301)

[2.2.3 Precondition Strengthening 3](#_Toc85019302)

[2.2.4 Postcondition Weakening 4](#_Toc85019303)

[2.2.5 Sequencing Rule 4](#_Toc85019304)

[2.2.6 Forwards and Backwards Reasoning 5](#_Toc85019305)

[2.2.7 Conditional Rule 5](#_Toc85019306)

[2.2.8 While Rule 6](#_Toc85019307)

[2.2.9 While Rule for Total Correctness and Termination 7](#_Toc85019308)

[2.2.10 Additional Equivalence Rules 7](#_Toc85019309)

[2.3 Dafny 8](#_Toc85019310)

[2.3.1 IDE 9](#_Toc85019311)

[2.3.2 Methods 12](#_Toc85019312)

[2.3.3 Pre- and Postconditions 12](#_Toc85019313)

[2.3.4 Assertions 13](#_Toc85019314)

[2.3.5 Functions 13](#_Toc85019315)

[2.3.6 Loop Invariants 13](#_Toc85019316)

[2.4 Teaching of Formal Methods 14](#_Toc85019317)

[2.4.1 Advantages of Tool Support 14](#_Toc85019318)

[2.4.2 Disadvantages of Tool Support 14](#_Toc85019319)

[2.5 Prior Art 15](#_Toc85019320)

[2.6 Program Verification Architecture 15](#_Toc85019321)

[2.6.1 Overview 15](#_Toc85019322)

[2.6.2 Verification Conditions 16](#_Toc85019323)

[2.6.3 Suggested Architecture 16](#_Toc85019324)

[3 Conclusion 17](#_Toc85019325)

[4 Project Tasks 17](#_Toc85019326)

[4.1 Project Deliverables 17](#_Toc85019327)

[4.2 Student Tool Development 17](#_Toc85019328)

[4.3 Educator Tool Development 17](#_Toc85019329)

[5 Project Plan 18](#_Toc85019330)

[6 Risks 19](#_Toc85019331)

[6.1 Time Management 19](#_Toc85019332)

[6.2 Lack of Expertise 19](#_Toc85019333)

[Bibliography 20](#_Toc85019334)

List of Figures

[Figure 1: Hoare Logic Notation {Gordon, 2016 #8} 3](#_Toc85035860)

[Figure 2: Hoare Triple Notation {Gordon, 2016 #8} 3](#_Toc85035861)

[Figure 3: A Hoare Triple 3](#_Toc85035862)

[Figure 4: Hoare Assignment Axiom Notation {Gordon, 2016 #8} 3](#_Toc85035863)

[Figure 5: Precondition Strengthening Notation {Gordon, 2016 #8} 3](#_Toc85035864)

[Figure 6: Example of Precondition Strengthening 4](#_Toc85035865)

[Figure 7: Postcondition Weakening Notation {Gordon, 2016 #8} 4](#_Toc85035866)

[Figure 8: Example of Postcondition Weakening 4](#_Toc85035867)

[Figure 9: Sequencing Rule Notation {Gordon, 2016 #8} 4](#_Toc85035868)

[Figure 10: Example of the Sequencing Rule [1] 5](#_Toc85035869)

[Figure 11: Backwards Reasoning [2] 5](#_Toc85035870)

[Figure 12: Conditional Rule Notation {Gordon, 2016 #8} 5](#_Toc85035871)

[Figure 13: If/Else Weakest Precondition Proof [2] 6](#_Toc85035872)

[Figure 14: While Rule Notation {Gordon, 2016 #8} 6](#_Toc85035873)

[Figure 15: Loop weakest precondition proof 6](#_Toc85035874)

[Figure 16: While Loop for Total Correctness 7](#_Toc85035875)

[Figure 17: Example of Total Correctness 7](#_Toc85035876)

[Figure 18: Variable Declaration Syntax 8](#_Toc85035877)

[Figure 19: Method Assignment Syntax 8](#_Toc85035878)

[Figure 20: Example of Method Assignment {Smith, 2021 #9} 8](#_Toc85035879)

[Figure 21: Features of the Dafny IDE 9](#_Toc85035880)

[Figure 22: Initial and Current IDE architecture (Arrows indicate data transfer, red arrows indicate diagnostics) [5] 10](#_Toc85035881)

[Figure 23: Dafny IDE colouring Style 11](#_Toc85035882)

[Figure 24: Example of re-verification. Between snapshot 0 to 1 only the Bar method is re-verified. Between snapshot 1 to 2 all entities need to be re-verified as all entities are dependent on the P function 11](#_Toc85035883)

[Figure 25: A method in Dafny [3] 12](#_Toc85035884)

[Figure 26: A postcondition in Dafny [3] 13](#_Toc85035885)

[Figure 27: A function in Dafny [3] 13](#_Toc85035886)

[Figure 28: A loop invariant in Dafny [3] 14](#_Toc85035887)

[Figure 29: BVD Architecture 16](#_Toc85035888)

[Figure 30: BVD GUI 16](#_Toc85035889)

[Figure 31: ESC/Java Architecture 17](#_Toc85035890)

[Figure 32: Why3 GUI {Jean-Christophe Filliâtre, 2013 #12} 17](#_Toc85035891)

[Figure 33: A System to Check Proofs {Gordon, 2016 #8} 18](#_Toc85035892)

[Figure 34: Example of Verification Condition 19](#_Toc85035893)

# 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 grow. 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 rational 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 will assistance for formatting their proofs checking that their proofs are syntactically and semantically correct, 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 the educators.

## Purpose

The purpose of this project is to develop a tool to support in the teaching of formal methods using the Dafny programming language.

## Goals

The goal is to develop a tool support that provides the following functionality:

* Implement a function called wpp into Dafny, that inputs weakest precondition logic to be verified, like the assert function in Dafny
* 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
* Verify the correctness of the weakest precondition proof entered
* Provide output as to what wpp logic is correct and why it may not be correct

## 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, so 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 will 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 authentication may be required. Essentially the scope, of this project is anything that can be included in a VSCode extension to assist in the tools creation.

# 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.

## Weakest Precondition Proofs and Hoare Logic

A formal proof of a statement is a combination of lines culminating with the statement, such that each line is an example of an axiom or continues from a previous line by a rule of 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 inference rules can be specified with the notation in Figure 1.

Figure 1: Hoare Logic Notation [1]

The demonstrates that the conclusion can be proved from which are hypotheses. [1]

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 [1]

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. An example of a valid Hoare triple can be:

Figure 3: A Hoare Triple

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. [2]

### 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 [1]

### Precondition Strengthening

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

Figure 5: Precondition Strengthening Notation [1]

An example of this can be seen as below:

Figure 6: Example of Precondition Strengthening

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 backwards reasoning.

Figure 7: Postcondition Weakening Notation [1]

An example of this can be seen below:

(assignment axiom)

(pure logic)

(precondition strengthening)

(laws of arithmetic)

(postcondition weakening)

Figure 8: Example of Postcondition Weakening

In the above example is included in the reasoning, making the specification weakener, even if it adds no meaning when it is multiplied by .

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

### 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 [1]

An example of this can be seen below:

(i)

(ii)

(iii)

Therefore, (i) and (ii) can be sequenced.

(iv)

Furthermore, (iv) and (iii) can be sequenced.

(v)

Figure 10: Example of the Sequencing Rule [1]

### 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:

Figure 11: Backwards Reasoning [2]

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 . [2]

### 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 follow the following notation:

Figure 12: Conditional Rule Notation [1]

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

Figure 13: If/Else Weakest Precondition Proof [2]

Here the weakest precondition is when the postcondition is is propagated up through the code. The proof in Figure 11 begins with the postcondition before being passed into the bottom of each if/else branch before joining back up again before the if else statement with the and junction as both branches are possible and the conditions for both branches must be met. To formulate the weakest precondition proof further equivalence laws can be used to simplify the expression.

### 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 they 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 [1]

An example of a loop invariant can be seen in Figure 12 below, and follows the notation from above:

(Precondition Strengthening)

(Simplifying)

Figure 15: Loop weakest precondition proof

### 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. A brief example can be seen below:

is

Figure 17: Example of Total Correctness

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

An example of this is shown below using the Multiply method:

Figure 20: Example of Method Assignment [3]

## Dafny

Dafny is a programming language with the ability to statically verify the correctness of programs with respect to relevant specifications determined in the written program. Dulled down, does the program do what the programmer intended. The programmer then just needs to write bug-free annotations, often easier than writing the code. [4]Its main purpose is to verify the functional correctness of programs and act as an interactive theorem prover. It is imperative, sequential, supports generic classes, inheritance and abstraction, methods and functions, dynamic allocation, inductive and co-inductive datatypes, and specification constructs. Verification occurs when the specifications match the code. These more important specifications include preconditions, postconditions, frame specifications such as read and write, and termination. More specifications include ghost variables, recursive functions, and data types such as sets and sequences. All specifications and ghost variables are applied only in verification, which is a part of the compiler but are omitted during execution. Ultimately, when the language produces verification errors the programmer can change the programs type declarations, specifications and statements just like a static type checker. [5]

### IDE

Program verifiers utilize three major integrated subsystems. Firstly, is the logic it uses, such as Hoare-style program logic or type theory, secondly is the automation mechanism such as decision procedures or proof search strategies. These 2 subsystems make up the proof system and affects how the programmer interacts with verification system through the manipulation of the input language. Thirdly, is the languages integrated development environment (IDE), which attempts to reduce the understandability of the proof system, making modifications simpler for the programmer. Dafny’s most developed IDE is an extension in Microsoft Visual Studio / Visual Studio Code (VS/VSCode) [6].

#### Features

The IDE provides the following programming features for program verification assistance:

|  |  |
| --- | --- |
| Feature | Description |
| Continuous Processing | The program verifier is run in the background providing instantaneous feedback. |
| Non-Linear Editing | The buffer can be edited from anywhere. A change in the buffer will make the verifier to reconsider proof obligations. |
| Multi-Threading | The IDE makes use of available multi-threading hardware. These concurrent threads are adjusted dynamically dependent on the complexity of verification tasks. |
| Dependency Analysis and Caching | The IDE caches verification outcomes and computer dependencies. Before beginning a new verification task the system interacts with the available cache reducing the programmers wait times. This is essential when the programmer need to make many small variations to the program to achieve overall approval, hence fluid response times from the verifier is crucial. |
| Showing Information | The IDE makes addition information available regarding, induction schemes, types, loop invariants and syntactic shorthand’s. This is accessible to the programmer via a hovered text which appears after the user’s cursor is hovered over certain program text, reducing clutter in the text editor |
| Integrated Debugging | Error messages gathered during verification are displayed when the user hovers their cursor over an identified error. This occurs because of the integration of the Boogie Verification Debugger (BVD) into the IDE. |

Figure 21: Features of the Dafny IDE

#### Tool Architecture

The Dafny programming language is built above the Boogie verification engine. For continuous processing to occur the IDE Dafny extension sends the program to the verifier whenever a change is made by the user to the text buffer. The program then translates the correctness proof obligations into Boogie, which is an intermediary language used for program verification. Boogie contains several declarations, such as variables and procedures, that are needed to formalize programs in higher-level languages. For example, a method in Dafny is encoded to a Boogie procedure that details the conditions of the methods specification, and a Boogie procedure implementation that details the method body and verifies that the method specification matches. Striped down, each function written in Dafny is translated to a Boogie function and procedure implementation. The Boogie program is then sent to the verifier, a automatic reasoning engine known as the SMT-solver Z3, which generates diagnostics on every Boogie implementation. These program diagnostics are then circulated back to the VS extension IDE to display the verification diagnostics to the user. [6]

Diagram

Description automatically generated

Figure 22: Initial and Current IDE architecture (Arrows indicate data transfer, red arrows indicate diagnostics) [6]

The initial architecture provides responsiveness for the user when verifying smaller programs but does not scale well, when many functions and methods need to be verified in larger programs. Therefore, the architecture was changed to verify separate tasks in parallel, allowing multi-threading using .NET Task Parallel Library. Each task will output verification conditions taking advantage of multiple solvers in a dynamically allocated allotment of solvers, allowing the user to see error asynchronously. [6]

The Visual Studio IDE extension follows the ensuing procedures:

1. Visual Studio extension is notified of a new snapshot, i.e. a change to the text buffer.
2. The extension recomputes syntax highlighting, through a lexical scan.
3. After 0.5 seconds of idleness, the extension runs the parser, resolver, and type checker over the text buffer.
4. If these phases are passed without error, the information is passed on to the user in hover text.
5. The text buffer is then sent asynchronously to the verifier.
6. As verification errors become available, they are sent up to the IDE extension.
7. Once these steps are completed a new snapshot can be verified.

Users typically wonder if the verification has been completed. The user is made aware of the verification completion by a panel on the side bar in the bottom left of the screen that indications when the verification is occurring and when it is complete. Additional colors are added into the margins and scroll panel to display modification to the snapshot: green indicates that the new modification is verified, red indicates that the new modification is not verified, red arrows indicate any lines that may have been removed, and blue indicates that any other changes that may have been made. An example of these colours is shown in Figure 2. [6]

A screenshot of a computer

Description automatically generated with medium confidence

Figure 23: Dafny IDE colouring Style

#### Caching

The popular technique of caching improves the responsiveness of the IDE extensions. Re-verifying every snapshot would be computationally expensive, caching allows the verifier to store error diagnostics from previous snapshots so that the entirety of the new snapshot does not need to be verified, only the parts that are dependent on the entity affected by the new change. This allows fast feedback when verifying large programs. This technique relies on detecting changes to program entities and following the dependencies of these entities. Changes to program entities are detected by an entity checksum for every function, method and specification. The checksum is computed based on the Dafny abstract syntax tree, so any changes to comments will not trigger an entity change. Furthermore, changes to dependent entities are then detected by a dependency checksum based on its own entity checksum and any other entity it is dependent upon. Changes to the values of these checksums from the current snapshot to the cached data are used to determine what part of the program needs to be re-verified. [6]

Graphical user interface, text

Description automatically generated

Figure 24: Example of re-verification. Between snapshot 0 to 1 only the Bar method is re-verified. Between snapshot 1 to 2 all entities need to be re-verified as all entities are dependent on the P function

#### User Interaction

A verification system contains information on how verification conditions are determined. Dafny utilizes heuristics to display induction hypotheses. For example, a programmer may want to know the rules regarding which calls are recursive or co-recursive. The Dafny resolver and type checker can connects information with Dafny’s abstract syntax tree nodes. In the IDE extension, this additional information such as default decreases clauses is shown to the user as a hover text when the curser is hovered over an AST node, in this case the decreases node.

The IDE extension provides error reporting to the user, when verification fails, and the user must debug their program. This may occur due to a multitude of reasons and the user may need additional information to solve the problem. One debugging method involves adding assert and assume statements to ask the verifier additional information about the program without directly manipulating it. Additional information can be provided by analysing counterexamples provided by the solver, with the assistance of the BVD to make it human readable. When verification fails a red squiggly line indicates the reported error and its return path. The error information can also be shown as hover text, for example if a post condition violation occurs. The blue squiggly lines indicate warnings. [6]

### Methods

Dafny is an imperative programming language that contains the typical methods, variables, types etc. One of the most used entity in a Dafny program is a method. A method is a section of executable code and may be referred to as a procedure or function in other programming languages. A method follows the typical construct as shown in Figure 4. Here the method keyword is used followed by a declaration of ‘Abs’ which takes a parameter x of integer type and returns an integer r. Each parameter and return value needs to be specified with a type with ‘:’ after each name. In the body of code, entity assignments are declared as ‘:=’. Statements need to be followed by ‘;’. To return a value the return value names must be assigned the value the programmer wants to return, acting as a local variable, however the input parameters are only read-only. The if statement would act as you expect with a Boolean predicate deciding which branch to take. Return statements are declared using the return keyword with the value assigned to return put into r. [4]

Text, letter

Description automatically generated

Figure 25: A method in Dafny [4]

### Pre- and Postconditions

The annotations described above could be for any imperative language. Dafny’s influence comes from being able to specify the behaviour of these methods. A specification for ‘Abs’ in Figure 4 above for example would be that return value is always a positive value. Typically, this specification would be placed in a comment but how would a programmer know the program does what the comment specifies. With appropriate annotations Dafny can prove that the method is programmatically true. The most basic form of specification are pre- and postconditions. In the case of ‘Abs’ the return value always being positive is an example of a postcondition. The postcondition is declared by the ensures keyword in the methods declaration, between the return values and method body as shown in Figure 5 and entails a boolean expression. The Boolean expression must hold true after all instances of the method to be verified correct. Although chaining multiple postconditions are possible it is also possible and more desirable to declare multiple postconditions for debugging purposes. Verification errors will be thrown when either the annotations don’t match the code or the verifier Boogie isn’t ‘smart’ enough to prove the specifications which does not typically occur. The precondition is declared by the requires keyword in the methods declaration and entails a boolean expression. The Boolean expression in this case must hold true before all instances of the method to be verified correct. It must be said that not all methods need to have a precondition or a postcondition. [4]

Text, letter

Description automatically generated

Figure 26: A postcondition in Dafny [4]

### Assertions

Assertions are used midway through a method and uses the assert keyword and then followed by a boolean expression. The purpose of an assertion is to determine if its entailing Boolean expression holds when that part of the code is reached, similar to the pre- and postconditions. Assertions are a useful tool when debugging annotations, and the behaviour of local variables, by checking what is supposed to be true at certain parts of the code is true.

### Functions

A function in Dafny is similar in declaration to a method but cannot write to memory, contains only one expression and are required to a single unnamed return value as shown in Figure 6. In order to instrument the abs function, an if expression must be used, which acts like a ternary operator. The advantage of using a function is that they can be used directly in annotations, such as assertions. Generally, functions do have pre- and postconditions attached to them, however they don’t need to be defined.

Text

Description automatically generated with medium confidence

Figure 27: A function in Dafny [4]

### Loop Invariants

In Dafny there is no way to determine how many times a loop will occur, however Dafny needs to explore this path in the program. Do Dafny makes use of another annotation known as loop invariants, specified by the loop invariant keyword. A loop invariant holds true when the loop begins and after every loop execution is completed, representing invariant entities in the program. Loop invariants are a property reserved for each execution of the loop. Dafny verifies that the invariant condition holds upon entry and is maintained by the loop. Dafny can only analyse the loop body based on the loop invariant and the loop guard. i.e. loop condition and cannot determine this for itself. An example of a loop invariant can be seen in a while loop that increments variable until , as shown in Figure 7. It can be determined that as the variable is set before the start of the loop and only increments.

Text

Description automatically generated

Figure 28: A loop invariant in Dafny [4]

## 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. [7]

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. [8]

### 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. [7]

## Prior Art

### Boogie

Boogie is an intermediate verification language (IVL) that produces verification conditions for programs. It acts as a program verifier for several source languages and converts the verification conditions into logical formulas to be sent the verification conditions to theorem provers. Boogie can be used as both an input and output for specification interpretation and predicate construction. The theorem prover then verifies the correctness of the original program. A further explanation of Boogie is provided in 2.5.4 below. [9]

### Boogie Verification Debugger (BVD)

Software verification technology developed to a stage where programs written in intricate programing languages can be verified, through the use of tools. Although these tools are difficult to use. Essential the tools understand the programs, but we do not understand the tools. The Boogie Verification Debugger (BVD), assists users to understand the output given by program verifiers. This tool improves the communication capabilities between the verifier and the user. This allows the user to explore and understand failing verification states.

Verification tools differ in their level of automation. There are fully automatic verifiers which include interpreters and model checkers and bound their reasoning to restricted domains. There are also interactive verifiers that accept user input and provides many proof assistances to the user. These tools expose to the users to internal proof conditions. The BVD users a verifier that uses both automation and interaction to verify a program, known as auto-active verification. In an auto-active verifier the user can generate assertions to be proven. Interactivity is conducted in the program source language by declaring pre- and postconditions or loop invariants. Dafny, discussed in 2.3, is an example of an auto-active verifier.

Auto-active verifiers use the technique of translating a source program into an intermediate verification language (IVL) such as Boogie or Why3. This IVL generate verification conditions which are sent to a reasoning engine such as Z3, which is a Satisfiable Modulo Theories (SMT) solver. [10]

Diagram, timeline

Description automatically generated

Figure 29: BVD Architecture [10]

When the verification conditions are generated by boogie and the reasoning engine determines a failing state a counter example is provide for the user, in the form of a GUI such as the one shown below:

Graphical user interface, application

Description automatically generated

Figure 30: BVD GUI

### ESC/Java

The program checking tool ESC/Java uses the technology known as extended static checking (ESC) for Java programs. It is static because the program gets verified without it being run and its is extended because the catches more errors than type checkers. The ESC utilizes a theorem prover to formally reason the correctness of a program. ESC allows the user to specific design conditions using annotations and provides warnings if their specifications are violated. The system follows the following architecture shown below. [11]

A picture containing text

Description automatically generated

Figure 31: ESC/Java Architecture

### Why3

Another program verifier is Why3 and is platform designed for deductive program verification. It offers a powerful language for specifications called WhyML and utilizes external theorem provers to generate verification conditions. An example of its GUI is shown below. [12]

Graphical user interface, text, application

Description automatically generated

Figure 32: Why3 GUI [12]

### KeY-Hoare

TODO MAYBE

## 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 33: A System to Check Proofs [1]

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 34: 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. [1]

#### 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.

# 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 function called wpp into Dafny, that inputs weakest precondition logic to be verified, like the assert function in Dafny
* 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
* Provide output as to what wpp logic is correct and why it may not be correct

# Project Plan

GANTT CHART

# Risks

This project will be presented with several risks, however as this project has no practical 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 (XXX), and his fellow contacts. As Dr Smith is an expert in the field of formal methods with industry and education experience, the projects level of sophistication needed can be met.

# Bibliography

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

[2] 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.

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

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

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

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

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

Context of Software Engineering," 2014.

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

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

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

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

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

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

# 