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](#_Toc84867774)

[1.1 Topic 1](#_Toc84867775)

[1.2 Goals 1](#_Toc84867776)

[1.3 Scope 1](#_Toc84867777)

[2 Background 1](#_Toc84867778)

[2.1 Formal Methods 1](#_Toc84867779)

[2.2 Weakest Precondition Proofs and Hoare Logic 1](#_Toc84867780)

[2.2.1 Hoare Triples 1](#_Toc84867781)

[2.2.2 Assignment Axiom 1](#_Toc84867782)

[2.2.3 Precondition Strengthening 2](#_Toc84867783)

[2.2.4 Postcondition Weakening 2](#_Toc84867784)

[2.2.5 Forwards and Backwards Reasoning 2](#_Toc84867785)

[2.2.6 Sequencing Rule 3](#_Toc84867786)

[2.2.7 Conditional Rule 3](#_Toc84867787)

[2.2.8 While Rule 4](#_Toc84867788)

[2.2.9 While Rule for Total Correctness and Termination 4](#_Toc84867789)

[2.3 Dafny 5](#_Toc84867790)

[2.3.1 IDE 5](#_Toc84867791)

[2.3.2 Methods 8](#_Toc84867792)

[2.3.3 Pre- and Postconditions 8](#_Toc84867793)

[2.3.4 Assertions 9](#_Toc84867794)

[2.3.5 Functions 9](#_Toc84867795)

[2.3.6 Loop Invariants 9](#_Toc84867796)

[2.4 Teaching of Formal Methods 10](#_Toc84867797)

[2.4.1 Advantages of Tool Support 10](#_Toc84867798)

[2.4.2 Disadvantages of Tool Support 10](#_Toc84867799)

[2.4.3 Examples of Tool Supports 11](#_Toc84867800)

[3 Conclusion 11](#_Toc84867801)

[4 Project Tasks 11](#_Toc84867802)

[5 Project Plan 11](#_Toc84867803)

[6 Risks 12](#_Toc84867804)

[Bibliography 12](#_Toc84867805)

[Appendix 13](#_Toc84867806)

[6.1 Initial Thesis Topic Explanation 13](#_Toc84867807)

List of Figures

[Figure 1: Featurs of the Dafny IDE 2](#_Toc84503006)

[Figure 2: Initial and Current IDE architecture (Arrows indicate data transfer, red arrows indicate diagnostics) [3] 2](#_Toc84503007)

[Figure 3: Dafny IDE colouring Style 3](#_Toc84503008)

[Figure 4: 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 4](#_Toc84503009)

[Figure 5: A method in Dafny [1] 5](#_Toc84503010)

[Figure 6: A postcondition in Dafny [1] 5](#_Toc84503011)

[Figure 7: A function in Dafny [1] 6](#_Toc84503012)

[Figure 8: A loop invariant in Dafny [1] 6](#_Toc84503013)

# Introduction

## Topic

## Goals

## Scope

# Background

## Formal Methods

## 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

The demonstrates that the conclusion can be proved from which are hypotheses. {Gordon, 2016 #8}

### 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

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. {Krysta Yousoufian, 2021 #5}

### 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 Axiome Notation

### Precondition Strengthening

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

Figure 5: Precondition Strengthening Notation

An example of this can be seen as below:

Figure 6: Example of Precondition Strengthening

The precondition specification for is , so becomes .

### 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

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.

### 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 9: Backwards Reasoning {Krysta Yousoufian, 2021 #5}

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 . {Krysta Yousoufian, 2021 #5}

### Sequencing Rule

### Conditional Rule

The examples shown until now have only involved linearly executed statements. Although, if/else statements have multiple possible branches adding complexity to the Hoare triple. For example:

Figure 10: If/Else Weakest Precondition Proof {Krysta Yousoufian, 2021 #5}

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. This separates the initial statements, loop guard and step statement. 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. 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 loops execution. When writing a loop invariant, the programmer typically gets inspired by either the postcondition or the loop guard. An example of a loop invariant can be seen in Figure 12 below:

Figure 11: Loop weakest precondition proof

The loop begins with

### While Rule for Total Correctness and Termination

Most loops typically terminate. So when proving a loops correctness, a proof involving the termination metric needs to be computed. In Hoare logic the specifications only hold if a loop terminates. 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.

## 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. [1]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. [2]

### IDE

Program verifies 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) [3].

#### 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 12: Featurs 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. [3]

Diagram

Description automatically generated

Figure 13: Initial and Current IDE architecture (Arrows indicate data transfer, red arrows indicate diagnostics) [3]

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. [3]

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. [3]

A screenshot of a computer

Description automatically generated with medium confidence

Figure 14: 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. [3]

Graphical user interface, text

Description automatically generated

Figure 15: 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. [3]

### 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. [1]

Text, letter

Description automatically generated

Figure 16: A method in Dafny [1]

### 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. [1]

Text, letter

Description automatically generated

Figure 17: A postcondition in Dafny [1]

### 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 18: A function in Dafny [1]

### 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 19: A loop invariant in Dafny [1]

## 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. {Shaoying Liu, 2014 #6}

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. {Wolfgang Ahrendt, 2009 #7}

### 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. {Shaoying Liu, 2014 #6}

### Examples of Tool Supports

#### ESC/Java

#### Why3

#### Boogie

#### Spec

#### KeY-Hoare

#### Haskell

# Conclusion

* There are many formal verifiers that are built within a languages source code and uses WPP as a logic base
* No formal verifiers show logic or prove user WPP proofs at editor level
* As an educational tool a verifier at editor level would provide students with logically knowledge of what the verifiers are doing under the hood in a ‘hand-written’ context.
* Either write wpp logic directly into VSCode extension using Typescript or use an external SMT solver/engine
* The users wpp must follow Hoare logit axioms

# Project Tasks

BRIEF BREAKDOWN AND EXPLANTION OF TASKS

# Project Plan

GANTT CHART

# Risks

# Bibliography

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

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

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

# 

# Appendix

## Initial Thesis Topic Explanation

This project will develop VS Code extensions which allow users to write and check weakest precondition proofs on Dafny code. This will allow future students of CSSE3100/7100 to readily format their assignments, and for tutors of CSSE3100/7100 to readily check assignments for correctness. It will also provide a valuable tool for software developers wanting to understand the behaviour of complex code.

You must either have knowledge of the following, or be keen to gain it:

- Dafny programming language

- weakest precondition reasoning

- TypeScript or JavaScript