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](#_Toc84412006)

[1.1 Topic 1](#_Toc84412007)

[1.2 Goals 1](#_Toc84412008)

[1.3 Scope 1](#_Toc84412009)

[2 Background 1](#_Toc84412010)

[2.1 Formal Methods 1](#_Toc84412011)

[2.2 Dafny 1](#_Toc84412012)

[2.2.1 IDE 1](#_Toc84412013)

[2.2.2 Boogie and Z3 4](#_Toc84412014)

[2.3 Weakest Precondition Proofs 4](#_Toc84412015)

[2.4 Teaching of Formal Methods 4](#_Toc84412016)

[3 Project Tasks 5](#_Toc84412017)

[4 Project Plan 5](#_Toc84412018)

[5 Risks 5](#_Toc84412019)

[Bibliography 6](#_Toc84412020)

[Appendix 7](#_Toc84412021)

[5.1 Initial Thesis Topic Explanation 7](#_Toc84412022)

List of Figures

[Figure 1: Initial and Current IDE architecture (Arrows indicate data transfer, red arrows indicate diagnostics) [3] 2](#_Toc84430040)

[Figure 2: Dafny IDE colouring Style 3](#_Toc84430041)

[Figure 3: 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](#_Toc84430042)

[Figure 4: A method in Dafny {Jason Koenig, 2012 #3} 5](#_Toc84430043)

[Figure 5: A postcondition in Dafny {Jason Koenig, 2012 #3} 5](#_Toc84430044)

# Introduction

## Topic

## Goals

## Scope

# Background

## Formal Methods

## 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. |

#### 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 1: 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 2: 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 3: 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 4: 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 5: 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 6: A function in Dafny

### Loop Invariants

## Boogie and Z3

## Weakest Precondition Proofs

## Teaching of Formal Methods

# Project Tasks

# Project Plan

# 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