HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

COMPUTER SCIENCE

IFS INSTITUTE FOR SOFTWARE

Visual Studio Code Integration for the Dafny Language and Program Verifier

Bachelor Thesis

Department of Computer Science
University of Applied Science Rapperswil

Spring Term 2017

Author(s):	Rafael Krucker, Markus Schaden
Advisor:	Prof. Dr. Farhad Mehta
Project Partner:	Microsoft Research, Redmond, WA, USA
External Co-Examiner:	Dr. Valentin Wüstholz
Internal Co-Examiner:	Prof. Dr. Markus Stolze

Task Description – Bachelor Thesis
Visual Studio Code Integration for the Dafny Language and Program Verifier
FS 2017

1. Client & Supervisor

- *Dr. K. Rustan M. Leino, Microsoft Research, Redmond, WA, USA*
- *Client Contact:* leino@microsoft.com
- *Supervisor:* Prof. Dr. Farhad Mehta, HSR Rapperswil

2. Students

- Mr. Markus Schaden
- Mr. Rafael Krucker

3. Setting

Dafny is a language developed by Microsoft which offers built-in specification constructs. These include pre- and postconditions, frame specifications as well as termination metrics. Further support such as ghost variables and recursive functions are also implemented. Through such specification primitives, the Danfy verifier, invoked during compilation, can be used to verify the specified aspects of the functional correctness of a program.

Dafny is typically used via its Visual Studio [1] IDE integration under the Windows operating system. This integration allows for an efficient workflow of editing a program while constantly being given feedback about its the functional correctness. The Dafny compiler and verifier can additionally be invoked from the command line.

Microsoft would like to integrate of Dafny into the cross-platform Visual Studio Code [2] IDE. Work on this has already been started through a plugin by Jonathan Rionatan [3]. It currently works within the mono-environment [4] and provides feedback from the verifier.

4. Goals

The main goal of this thesis project is to improve the existing integration of Dafny within Visual Studio Code and thereby allow Dafny to be effectively used in a cross-platform setting. In particular, the following improvements and additions to the existing Visual Studio Code plugin are proposed:

1. Stable Working Release of the Plugin on the following Platforms
    - Windows 10 (.net-environment)
    - Linux (mono-environment)

    - MacOS (mono-environment)
2. Easy installation of the plugin, with an automated download of Dafny and the automatic setting of all system variables
3. Syntax-Highlighting
4. Compilation of Dafny Best Practices and reporting of their violations within the plugin
5. Automatic generation of contract/specification/manual proof suggestions for common and simple cases
6. Autocompletion for identifiers

Goals 1, 2, 3 and 4 have the highest priority since they provide the beginner with the greatest help.

Goal 5 is probably the most interesting feature, because it could bring much of the power of Dafny to the programmer with relatively little effort on his side. Since this feature does not have strong parallels to standard IDEs for programming, it will require thought and research to execute. Due to this, the focus of the project is currently planned here, after having learnt enough of the setting from the preceding goals.

Goal 6 currently has the lowest priority. It is unclear if autocompletion in the setting of Dafny is conceptually significantly different to IDEs for programming. The execution of this task is heavily dependent on the existing support from Visual Studio Code and the Dafny compiler, whereas its novelty and effectivity for the user is debatable to be currently placed higher in the list of priorities.

In addition to the goals stated above, the following points will be considered during the course of the project:

1. The use of Dafny in order to implement the features discussed.
2. Other currently unknown improvements to the workflow and IDE tooling.

5. Guidelines

The students and the supervisor will plan weekly meetings to check and discuss progress. The student will schedule meetings with the client as and when required (recommendation: 1 meeting per week of 1 hour duration).

All meetings are to be prepared by the students with an agenda. The agenda will be sent at least 24h prior to the meeting. The results will be documented in meeting minutes that will be sent to the supervisor.

A project plan must be developed at the beginning of the thesis to promote continuous and visible work progress. For every milestone defined in the project plan, the temporary versions of all artefacts need to be submitted. The students will receive a provisional feedback for the submitted milestone results. The definitive grading is however only based on the final results of the formally submitted report.

6. Documentation

The project must be documented according to the regulations of the Computer Science Department at HSR (see https://www.hsr.ch/Allgemeine-Infos-Bachelor-und.4418.0.html). All required documents are to be listed in the project plan. All documents must be continuously updated, and should document the project results in a consistent form upon final submission. All documentation and work artefacts have to be completely submitted in three copies on CD/DVD (one copy each for the client, university, and supervisor). Three printed copies of the report need to be submitted (one copy each for the client, external examiner, and supervisor).

7. Important Dates

Please refer to https://www.hsr.ch/Semesterdaten-2016-2017.13924.0.html.

8. Workload

A successful bachelor thesis project results in 12 ECTS credit points per student. One ECTS points corresponds to a work effort of 30 hours.

All time spent on the project must be recorded and documented.

9. Grading

The HSR supervisor is responsible for grading the bachelor thesis. The following table gives an overview of the weights used for grading.

Facet	Weight
1. Organisation, Execution	1/6
2. Report	1/6
3. Content	3/6
4. Final Presentation & Examination	1/6

The effective regulations of the HSR and Department of Computer Science apply (see https://www.hsr.ch/Ablaeufe-und-Regelungen-Studie.7479.0.html).

Rapperswil, 25.11.2016
Prof. Dr. Farhad Mehta

References:

[1] https://github.com/Microsoft/dafny/wiki/INSTALL

[2] https://code.visualstudio.com

[3] https://github.com/ferry-/dafny-vscode

[4] https://github.com/mono/mono

Contents

1 Abstract

The goal of this project is to integrate the Dafny programming language into Visual Studio Code. Emphasis is put into researching how Dafny programmers can be best supported during their work and how writing code can be made more productive.
Since Dafny offers built-in specification constructs, novel work is to provide tooling that makes using them easier. The most beneficial feature would be to implement a generic, context aware proof obligation generator that continuously suggests specification constructs to the programmer. This approach was eventually discarded because it was deemed unfeasible to implement after some research had been done. Instead, situations were identified that arise often during programming and specific aid with specification constructs was implemented for them. Another helpful feature is the displaying of counter examples where code written does not satisfy the corresponding specification constructs, allowing quick discovery of edge cases and refinement of specification constructs.
Next to language specific features, standard IDE mechanisms allow for great improvement regarding productivity. It was deemed paramount that the project implements the most common features such as go to definition and auto completion. This was achieved using the standard interfaces that Visual Studio Code provides, allowing programmers accessing these features in a well-established way.
It is of concern that new users can get started quickly, so that the user base continuous to grow. To support this, automatic installation on all major platforms was implemented. The installation resolves all dependencies such as the Dafny pipeline itself. To further maximize portability, the plugin implements the language server protocol. This allows for writing IDE agnostic language analysis platforms, making the plugin not only usable in Visual Studio Code, but integrable into some other IDEs with only minor adjustments.
This project concluded with the implementation of a production ready integration of Dafny into Visual Studio Code. The application of continuous integration allowed for a user base of about 300 people at the end of the project, proofing that the plugin is robust and works across multiple environments. Dafny programmers are supported in their coding not only with standard IDE mechanisms, but also Dafny specific features. Next to making the experience of programming more productive, this lays the foundation of a contentiously growing Dafny community.

2 Management Summary

Following the quickly growing digitalization of businesses and the therefore more complex applications being developed, two key points are gaining focus in today's IT landscape. The first one is the proven functional correctness of programs and the second one is the uprising of multi threaded applications. Writing multi threaded applications becomes much simpler once the functional correctness is proven, thus it can be stated that proven functional correctness is a stepping stone towards the easier implementation of parallelism. Dafny is a programming language which tries to move the focus of writing correct code towards writing correct specification constructs, which is often easier. Applying this concept consistently should result in being able to turn business requirements into correct working implementations quicker as with traditional languages. The proven correlation between the time of discovering an error and the cost of fixing it also strongly advises writing software which is proven correctly as early in the life cycle of the product as possible. Even though using Dafny in business is compelling for these reasons, its usage is still not widespread, something which this project tries to change.
The wide spread usage of a tool for programmers is mainly dictated by two factors, namely the burden of getting it to run and the support that it is able to offer the programmer.
To address the first point, the plugin was developed for Visual Studio Code, an IDE running on all major platforms. It was also given an installation routine which resolves all dependencies on all platforms automatically after pressing one button. This also allows programmers that are not that familiar with the console to rapidly develop Dafny programs, something that was not possible given the tooling existing up until now.
Regarding the second point, it was always paramount to offer as much help as possible to the programmer. This was done in two steps, the first one being implementing standard features that a programmer is used to when working with an IDE, which could be achieved within this project. The second step are language specific features. To offer these, some research was done in what situations often arise when programming Dafny in order to reveal which features a programmer could most benefit from without solving all specification construct suggestions in a generic way. This pragmatic approach helped the project stay in scope while still offering rich help in many common programming contexts.
While at first not a key concern, it was decided to implement the plugin according to an emerging standard in semantic language analysis platforms. This means that the plugin does not only work well with Visual Studio Code, but can be integrated into many other IDEs such as Emacs with very little adjustments, further broadening the possible Dafny user base.
During the project, production quality was always striven for, so that the end product was not a prototype which no one uses. Due to the application of continuous integration the user base of the plugin has already reached 300 people at the time of this writing, proving the usability and robustness of the product developed by this project. The project remains open source, inviting other to continue the work and share the benefits of Dafny with even more people.

3 Outline

3.1 The problem and its setting

This chapter presents the background of the project, the problem and its significance.

3.1.1 Introduction

Dafny is a language designed and implemented by Microsoft Research. It offers built-in specification constructs. These include pre- and postconditions, frame specifications as well as termination metrics. Further support such as ghost variables and recursive functions are also implemented. Through such specification primitives, the Dafny verifier, invoked during compilation, can be used to verify the specified aspects of the functional correctness of the program.
Dafny is typically used via its Visual Studio IDE integration under the Windows operating system. This allows for an efficient work flow of editing a program while constantly being given feedback about its functional correctness. The Dafny compiler and verifier can additionally be invoked from the command line.
Microsoft would like to integrate Dafny into the cross-platform Visual Studio Code IDE. Work on this has already been started through a plugin by Jonathan Rionatan. It currently works within the mono-environment and provides feedback from the verifier.

3.1.2 Statement of the problem

This thesis aims to research on how Dafny programmers can be best supported during their work and incorporate these findings in a production quality plugin for Visual Studio Code.

3.1.3 Significance of study

Standard programming techniques are beginning to show their limitations as multi-core and multi-threaded applications are becoming more and more popular, which are difficult and error prone. Proving functional correctness has the potential of helping the programmer construct reliable programs. Sadly, the use of this technology is not widespread yet. Providing better tool support has the potential of improving this situation. Here lies the significance of this project.

3.1.4 Scope and delimitation

The plugin is limited to be used in three defined environments, although they compromise a huge percentage of environments used in programming. The plugin offers a fixed set of features which are detailed in this thesis, but remains open for adaption and extension.

4 Motivation

This section first explains the main goal of this project and follows up with a section about current solutions for this problem setting and their shortcomings.

4.1 Main Goal

The main goal of this project is to make Dafny accessible for a wider user base. In order to this, the focus was laid on two main objectives.
The first one is to provide a simple setup for all the tooling that is necessary to program with Dafny. This includes, next to the IDE itself, also the compiler and the proof engine pipeline which Dafny uses, among other things. The central point of handling the setup does not only make the life of the programmer easier, but also allows for the control of upgrades and dependencies in an uniform manner.
The second objective is to support the programmer when actually writing code. Programmers are used to get rich support from modern IDEs and the question of choosing of a programming language for a project is often interwoven with the quality of tooling behind that language. To make Dafny attractive for a broader range of users, the IDE should offer Dafny specific features such as help with writing specification constructs. Through this, programmers are best able to learn quickly which features Dafny offers and how to use them. Next to the language specific features, standard IDE features which programmers have grown accustomed to must not be forgotten. If a new IDE does not offer support for features such as go to definition, find references, and a minimal set of refactorings, many programmers may stop using it after a short amount of time because the IDE does not support the work flow they have gotten used to.
The main goal of this project therefore lies in providing robust, well working solutions regarding these two objectives.

4.2 Current Solutions

This chapter details what shortcomings current solutions in this area have and where there is room for improvement. Rather than offering a checklist of each solution in regard to its features, another approach is chosen in this chapter. Important concepts of an ideal solution that satisfies the main objectives detailed in 4.1 are listed, combined with how most current solutions perform regarding the concept. When an existing solution stands out in some way, it may be named specifically.

4.2.1 Platform Independence

With three well established operating systems being used by different programmers, it is not feasible to have a solution that only works on one platform. There is often a trade off in this area regarding using powerful platform specific APIs and having a portable solution. While many other languages are supported by rich platform independent IDEs such as Eclipse [Ecl17] or those provided by JetBrains [Jet17], the current solutions for Dafny still lack in

this area.
Most IDEs that support Dafny only work correctly on Windows, Emacs [GNU17] with it's Dafny plugin is the only solution that works across all platforms. Emacs, while being a heavily used IDE, is an IDE with a very special methodology that does not suit all programmers, narrowing the range of people that can be reached by a Dafny integration. Further there are some cross platform IDEs that offer support for Dafny, for instance the old Dafny plugin for Visual Studio Code, but with those the integration itself does not work across all platforms. This evaluation shows that there is need for a truly platform independent Dafny integration in a well established IDE that has a wide user base.

4.2.2 Setup

This is an area where all current solutions lack comfort. Next to the plugin for the specific IDE, the user must also make sure to install the whole Dafny platform and configure the plugin correctly to make use of it. This is usually done either by editing a configuration file or by using a dialog in the IDE. Next to this being error prone and cumbersome for the user, this process is also dangerous when the IDE and the Dafny platform are further developed. Version updates may introduce breaking changes which will make it unable to work with a plugin if it is not well maintained.
Next to an automatic installation of all components that are necessary, it is important to have some integration tests in place that notify in case of breaking changes, something that is very difficult if the gathering of all dependencies is not done in a single place.

4.2.3 Usability

When designing a plugin, one usually does not have many options regarding usability, since the user interacts with the IDE rather than with the plugin. It is therefore important to make usage of the correct mechanisms that an IDE offers, for instance displaying compiler errors in the appropriate window or underlining warnings with the color the IDE uses for warnings also in other languages. When further user interaction is needed, for instance for the application of a refactoring or displaying configuration possibilities of a plugin, IDEs usually offer an idiomatic way to do this.
The existing solutions do a nice job in this area, if information of the plugins is displayed, it usually is done so using the correct mechanism that the IDE offers for that type of information. Almost all existing solutions sadly only display a subset of the information that the Dafny platform could provide, here new solutions could offer improvements. The exception is the existing Visual Studio integration of Dafny [Mic17e], which offers almost a complete interaction with the Dafny platform.

4.2.4 IDE Independence

All existing solutions are hardwired to an existing IDE. This means that when a plugin is written for Visual Studio and one for Emacs, all features have to be copied and implemented anew. This makes it cumbersome to widen the user range of Dafny users. It also makes the process of updating plugins error prone and asymmetric, since changes have to be done in two places.

When looking at the process of programming a plugin in an abstract way, it comes down to integrating a semantic language analysis framework with an existing IDE. There have been several attempts to unify the way this integration is done, with Microsoft weighing in with a solution for this they call the language server protocol [Mic17c], which is quickly gaining traction. When this protocol matures and a solution is written in terms of such the protocol, one can offer plugins for many different IDEs while writing the core logic only once. There exists great potential in following this strategy.

4.2.5 Feature Richness

Once solutions are installed and running, the most important thing is feature richness. Next to standard IDE features, which are a must and expected by programmers, language specific features are what really make a solution stand out. This especially is true for Dafny, since it offers an almost unique and very valuable approach to specification constructs. In this area, most existing solutions perform poorly. Of the very rich information about a program that Dafny provides, only a fractions is presented to the user.
A notable exception to this is the existing Visual Studio integration. It offers virtually all possibilities that Dafny provides in its GUI. It even integrates counter examples for failed proofs and a debugger. While most existing solutions do not provide many features, the Visual Studio integration is the example that new solutions should try to match regarding feature richness.

5 Preliminary Studies

This chapter documents some background towards specification construct generation and the proof theory behind it. The work done in this chapter was done towards the goal of generic specification construct generation as detailed in B.4.4. For reasons detailed in A.4 and insights won during the investigations listed in this chapter the implementation of that feature was deemed unfeasible. Nevertheless, the research lead to a deeper understanding of what is possible regarding specification construct generation and laid the basis for many other implemented features. Because the items researched in this chapter were not later implemented as is, the reading of it is not mandatory in order to understand the work ultimately done, but delivers some background on why the project took the final form it has.

5.1 Common problems when programming

Since Dafny offers built-in specification constructs, a programmer would greatly benefit from generation of contracts for common situations. This chapter first introduces three examples in programming, that could be made safer through the use of contracts. The solution is then generalized in order to be more widely applicable.

5.1.1 Example 1: Array access

Problem: A method accesses an array with an index, which is given as a parameter. The array may be a field or also be a parameter. The array may be null or the index may be out of bound.

Solution: Generate a precondition which checks if the array is not null and the index is in bound of the array.

```
1  method FindUsafe(a: array<int>, key: int) return (element: int)
2  {
3          return a[key];
4  }
5
6  method FindSafe(a: array<int>, key: int) return (element: int)
7          requires a ≠ null ∧ 0 ≤ key < a.Length
8  {
9          return a[key];
10 }
```

5.1.2 Example 2: Simple domain specific constraints

Problem: A method that processes withdrawals from a bank account must not make a bank balance negative.

Solution: Generate pre- and postconditions on methods which modify relevant fields, according to domain specific constraints.

```
class BankAccountUnsafe {
        var balance: int;

        method withdraw(amount: int) modifies this {
                balance := balance - amount;
        }
}

class BankAccountSafe {
        var balance: int;

        method withdraw(amount: int)
                requires balance ≥ amount
                ensures balance ≥ 0
        modifies this {
                balance := balance - amount;
        }
}
```

5.1.3 Example 3: More complex Domain specific constraints

Problem: A factory wants to model its processes. Their services consist of refining certain raw materials, which can interact aggressively with their machines. They have two types of machines, some which are subject to abreason over time, but also others which are very expensive and should not come into contact with aggressive materials. They want to make sure no aggressive materials come in contact with expensive machines under any circumstances.

Solution: Generate pre- and postconditions on methods which modify relevant fields, according to domain specific constraints.

```
class RawMaterial {
        var abreasesMachines: bool;
}

class NormalMachine {
        var prestine: bool;
        constructor() modifies this {
                this.prestine := true;
        }
        method refineMatieral(material: RawMaterial) {
                ...
        }
        method processMaterial(material: RawMaterial)
                requires material ≠ null
        modifies this {
                this.refineMatieral(material);
                this.prestine := ¬material.abreasesMachines;
        }
}

class ExpensiveMachine {
        var prestine: bool;
        constructor() modifies this {
                this.prestine := true;
        }

        method refineMatieral(material: RawMaterial) {

        }
```

```
30
31          method processMaterial(material: RawMaterial)
32                  requires material ≠ null
33                  requires ¬material.abreasesMachines
34                  ensures prestine
35          modifies this {
36                  this.refineMatieral(material);
37                  this.prestine := ¬material.abreasesMachines;
38          }
39  }
```

5.2 Concepts in proof theory

All three examples have in common, that without the correct preconditions, they should result in proof obligations which cannot be proven. This subsection first details three common concepts that occur when reasoning about proof obligations. The next subsection sets them into connections with the problems mentioned in 5.1.

5.2.1 Application of partial functions

One of the three problems can be expressed as the application of partial functions, which are defined by the following three objects:

- A set A called the input set of the function
- A set B called the output set of the function.
- A rule f that transforms some elements of A to some elements of B such that no element a from A is transformed to more than one element of B.[KK12, p. 197]

The definition states that not all input values may be mapped by the function. The problem here therefore is to ensure that the function is applied on only a valid subset of A.

5.2.2 Invariants

An Invariant can be defined as follows: A quantity which remains unchanged under certain classes of transformations. Invariants are extremely useful for classifying mathematical objects because they usually reflect intrinsic properties of the object of study.[Hun99, 282ff] An Invariant is therefore extremely useful when one wants to ensure certain conditions of an object, which must hold at all times. If an invariant is to be applied to an object with multiple attributes, it is usually defined as a postcondition on all attributes.

5.2.3 Non provable Goals

These are situations that are impossible to prove, because some postconditions do not hold or not sufficient information is available. They are very hard to detect and isolate from provable goals, although some work has found solutions under certain restrictions, for instance [Brü05]. When an unprovable goal is encountered in a context, it is much easier to simply state it as a precondition for the context to be valid, thus burdening the calling context with ensuring that the goal holds.

5.3 Concrete Application

This subsection finally shows how the patterns detailed above can be used to solve the programming examples, thus allowing generic solutions for many similar problems.

5.3.1 Resolving Bound checks

The situation shown in 5.1.1 is a very common situation while programming, basically one wants to prove that the index is always in the bounds of the array. Accessing an element in an array is an example of the Application of partial functions, where the set A only goes from zero to the length of the array minus one. Therefore one would have to prove that the application of the partial function does not result in an invalid element being given as an argument. The expression, which is used to get access, can be arbitrarily complex. The computation to ensure the in-boundness can be very expensive.
Also the second pattern discussed in Invariants could be applied, always ensuring that a given parameter is in bound of an array of an object, although this would not work with how invariants are normally applied, namely as postconditions. Since the expression that generates the index has nothing to do with the object itself, it is questionable if this is the right pattern to apply to this situation.
The third pattern, discussed in Non provable Goals, works very well if one assumes that it cannot be proven that the expression will always lead to a successful application of the partial function (although it could be proven in many cases). This allows to define this property as a precondition of the method, thus shifting the burden to the caller to always check his arguments. This is an easy and feasible solution to the first problem.

5.3.2 Enforcing Invariants

The example in 5.1.2 is an example of a domain specific limitation where a bank account's balance should never fall below zero. In this specific implementation the usage of Application of partial functions could be discussed, since it is implemented as a binary minus function which only allows subtrahends from a certain range, although this approach could not be used for all possible implementations and is therefore not a feasible solution.
The second pattern discussed in Invariants best describes the semantics of the situation in a very general way. It could simply be stated as an invariant, that the balance has always to be positive. This does not suffice though, as it does not isolate the parts yet which could break the invariant.
The third pattern, discussed in Non provable Goals, works very well in conjunction with the second one. All sub goals that should hold so that the invariant holds, in this case that the amount should be smaller than the balance, can be viewed as unprovable goals in this context. They can therefore be formulated as preconditions such that the caller has the burden of applying correct arguments to the function. This practice isolates the parts which could break the invariant, allowing to write the function as safe as possible.

5.3.3 Enforcing Complex Invariants

The example in 5.1.3 is also an example of a domain specific limitation, although more complex. It combines several classes together, which could also potentially be subtyped. The goal is to never let an expensive machine be subjected to abreason, therefore only allowing non-aggressive raw materials as input. In this specific implementation the usage of Application of partial functions could work very well, since the input set of the partial function is very small, namely only the value true on the abreasesMachines property of a raw material. However, if we extend the hierarchy of materials and implement the calculation of abreasesMachines differently, it is unclear if all cases could be computed efficiently.
The second pattern discussed in Invariants best describes the semantics of the situation in a very general way. It could simply be stated as an Invariant, that the pristine property on the expensive machine is always true. This does not suffice though, as it does not isolate the parts yet which could break the invariant. This is the same situation as in Enforcing Invariants
The third pattern could be used much the same way as in Enforcing Invariants. The condition, that the raw material may never abrease machines, can be written as a precondition and together with the usage of invariants holds the greatest amount of security regarding the domain constraints.

5.4 Conclusion

As the discussion above shows, all of the three problems can be solved through the application of Invariants and Non provable Goals. They make it unnecessary to solve the problems of Application of partial functions, which is often harder to do. All occurrences, where such a computation would be needed, can be seen as non provable goals and stated as preconditions for a method. The usage of invariants offers a syntactical transparent way of describing domain specific rules, and the implementation is a relative simple one, as it is simply translated into postconditions for all methods of an object. Together these two techniques offer solutions to many different problems in computer science, since they operate on a high abstraction, while still being syntactical transparent.
In the first example, the language itself has enough domain knowledge in order to generate the unprovable proof obligations, since it knows about the array type and its restrictions. In the two other cases the language needs more domain knowledge in order to generate the unprovable proof obligations. As was shown above, invariants are a good way of providing this domain knowledge. Once the unprovable proof obligations can be found, they can be used as hints for preconditions that can be suggested to the programmer.
The hardest part of the implementation is the identification of non provable subgoals that are in relation to an invariant. To do this, detailed knowledge has to be available of the control flow of a program and all possible outcomes of a computation have to be considered, although the problem can be relaxed if one allows for false positives in the identification of non provable subgoals. Since the plugin only offers refactorings and does not apply them automatically, the programmer still can decide if the setting of the non provable goal as precondition is necessary.

6 Results

This section details what steps were taken to reach the goal of this project as defined in 4.1. The steps are grouped into three parts, according to the objective they satisfy. They are listed in no particular order. The chapter details the features in a conceptual way, insight into the implementation can be found in the architecture documentation paper, which accompanies this paper.

6.1 Setup

This chapter explains features which were not implemented as the project was started. These features are mainly chosen in order to grow the community, as offering features as making starting coding with Dafny easier and supporting a wide range of IDEs accomplishes this goal.

6.1.1 Language Server

What is a Language Server A language server allows to integrate features like auto completion, go to definition, find all references in an easy way into an IDEs. Such a server mostly coexists with a client, which is a normally a native IDE plugin, which does the customization of the GUI, registers shortcuts or adds menu items. The language server provides a list of supported actions, which are normally shown in the IDE. This makes it very handy to extend it as well. The protocol between these two components is standardized by Microsoft, with the idea behind it to achieve IDE independence.

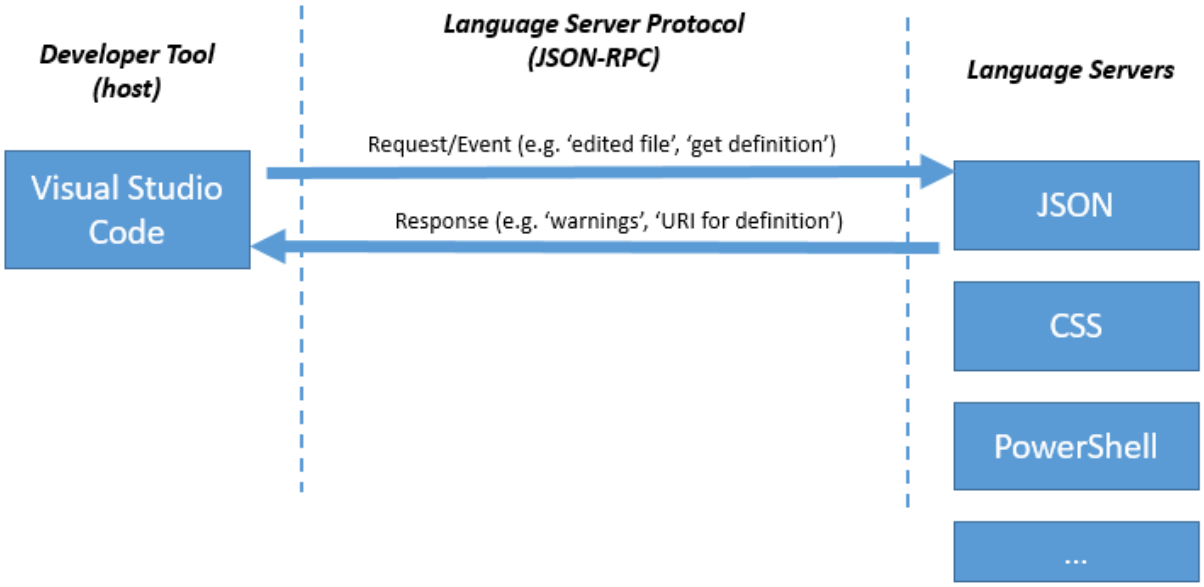

Figure 6.1: Language Server

Why were they chosen to implement Since Dafny is already supported on different operation systems, it was decided to make the plugin as platform and IDE independent as possible. Because Visual Studio Code supports the Language Server Protocol and is the reference implementation, the existing plugin was refactored to work as a language server. The strict splitting of the plugin into a client and a server had also the benefit, that the architecture had to be changed, which resulted in a much better and clearer structure.
In the future another IDE could just integrate the existing language server, make smaller tweaks to the client and it would work. No rewriting of the whole logic is needed, as one just can use the server, thanks to the standardized protocol.

What benefits do they entail Developing the plugin as a language server has the big benefit, that it could be easily integrated into existing IDEs, which support the Language Server Protocol, without reprogramming everything. Supporting auto completion, go to definition, find all references, would just work, without writing a single line of code. The only step is to start the Language Server and connect to it. Only the client needs to be adjusted, depending on the requirements of how the GUI should look like and probably the programming language to developing the client will change. See 7.1 for a overview of IDEs which were looked at and tried to be supported.

6.1.2 Automatic Installation

What is Automatic Installation Automatic Installation is the process to setup the whole Dafny environment in the background. Downloading the latest release from GitHub, extract it and setting all necessary configuration properties correctly is its main task. This process is platform dependent, because there is a zip file for each operating system.

Why was this chosen to implement The setup to start coding with Dafny was quite complicated. The whole installation was not straightforward. First one had to download the Dafny.zip from GitHub, which is hidden under releases. Second it had to be extracted into a directory. Afterwards installing the Dafny plugin for Visual Studio Code and setting the path to the DafnyServer.exe had to be done. Finally after a restart of Visual Studio Code it maybe worked.
Because this is quite complicated, especially if one wants to grow the community, this process must be much easier. For this reason it was decided to implement this important feature. The goal was that a new user can start coding in under 1 minute.

What benefits do this entail More people can start writing Dafny programs, not worrying about having to set up Dafny, struggling with configurations or not finding the release on GitHub. Also professors can integrate Dafny into their lectures, as long as students can install Dafny without problems.

6.1.3 Automatic Upgrade

What is Automatic Upgrade Automatic upgrade makes sure that always the latest version of Dafny is used. If a new release is published on GitHub this feature will check that,

and notify the user that there is newer release available. If the user wants to upgrade the Dafny environment, the feature 6.1.2 will take over.

Why was this chosen to implement Most people will never look at GitHub to check if there is a newer version of Dafny available. Especially if there is no need to, because everything can be installed automatically. Out of this reason, there needs to be a version check, which informs the user if there is a newer version.

What benefits do this entail Newer versions bring important bug fixes and new features, which would not be used if there is no version check. Users can install newer version the same easy way, as installing the environment the first time.

6.2 Language Agnostic Features

This chapter details the language agnostic features that were implemented during the project. The implementation was heavily guided by the language server protocol and state of the art IDE integrations of well established languages. With these features, the main goal was determining what programmers have become accustomed to when using an IDE and which features bring the biggest improvements regarding productivity. To get programmers to look deeper into the features that Dafny offers, it is important common tasks like navigating in a code base just work, because otherwise interest to learn new things fades quickly.
This chapter simply aims to explain why certain features were chosen and what improvements they bring to the daily routine of a programmer.

6.2.1 CodeLenses

What are CodeLenses CodeLenses in Visual Studio Code are a feature which is also common to many other IDEs. The idea is to display meta information about certain pieces of codes, for instance classes and methods. In Visual Studio Code this is done by adding an additional line of text to the editor wherever a codeLense should be placed.

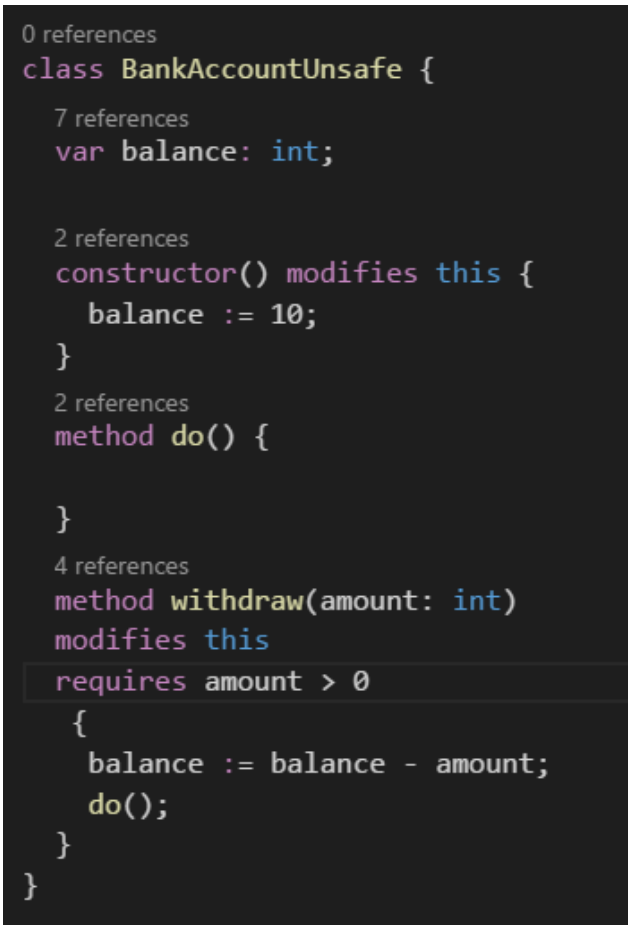

Figure 6.2: Code Lenses used with Dafny

When given locations of the references, it is possible to let Visual Studio Code highlight them in the preview window which opens when a codeLens is expanded. Visual Studio Code also groups references according to the file path in the location, so the programmer gets to see a map of all references ordered by containing file to the right of the preview window and can quickly navigate to them.

Why were they chosen to implement Since codeLenses can be used to quickly gain a deeper understanding of a code base, it was decided to integrate this feature. Another reason was that it is widespread in different IDEs, so that programmers have become accosted to it.
The first decision to be made was for which elements in the code codeLenses should be displayed. The trade off here is to provide enough information to work comfortably with the code base and not to clutter the workspace with codeLenses. It was decided to display codeLenses for classes, methods (including constructors) and fields, since they tend to have a wide scope in the code bases
A second consideration was which information should be displayed in a codeLens. When codeLenses are language specific, references and usages of the element are usually displayed. Since this allows the programmer to gain a deeper understanding of control flow and regions

affected by refactoring, it was decided to display this information also for the Dafny plugin. CodeLenses also allow commands to be executed when clicked upon, a logical conclusion is to implement go to reference when a reference in a codeLens is clicked.

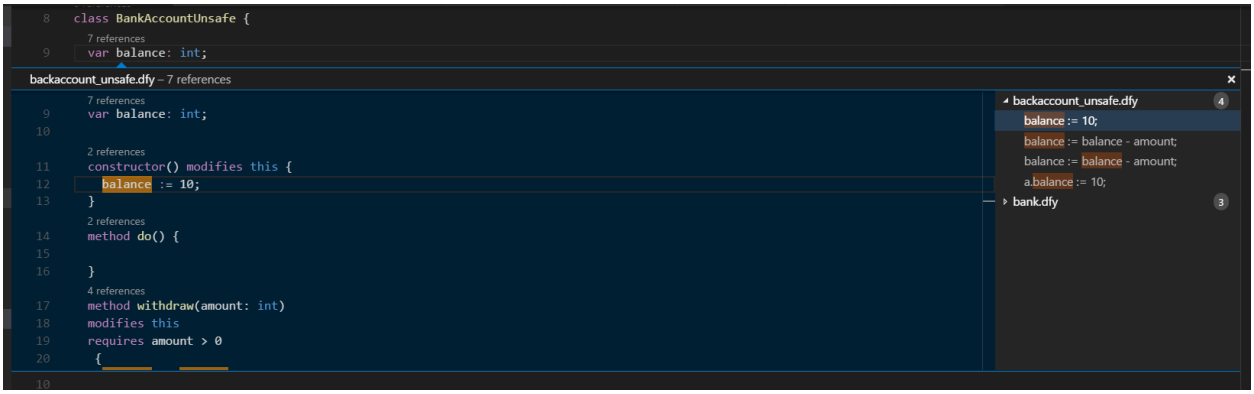

Figure 6.3: Expanded CodeLens showing the references to the field balance

What benefits do they entail CodeLenses allow to gain understanding of a code base rapidly. Scoping is visible at a glance, making it excellent to find out which regions of code are affected by a refactoring or how many classes rely on another class. By clicking on the references that are listed, one can also navigate the code base with ease, making it easy to follow program flow.
The existing solutions do not yet implement code lenses at all, a feature which is wide spread in other established language integrations. Once a programmer is used to working with code lenses, it usually becomes a feature that is quite heavily used and bring a big improvement in productivity, since relationships do not have to be recorded in a mental model. It stands to reason that this feature brings the biggest improvement in understanding control flow and navigating a code base when compared to other stand alone features, making it an important part of the plugin.

6.2.2 Code Completion

What is Code Completion Code completion prevents the programmer having to type out every identifier fully. Microsoft calls this feature IntelliSense in its products. Usually, when the programmer starts typing, a little popup appears in which the programmer can choose options that complete the code he is currently writing. The suggestions are usually context aware, because a popup that is cluttered with every identifier in the file currently opened does not bring an improvement in productivity.

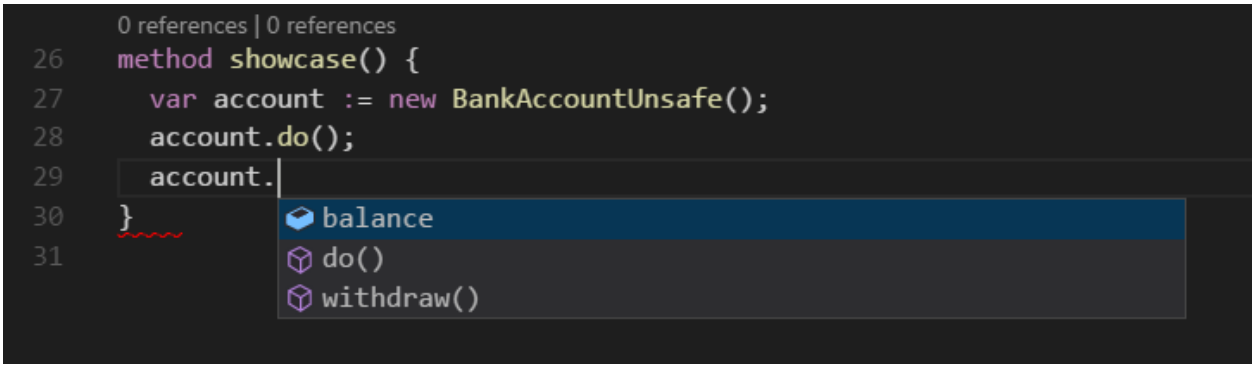

Figure 6.4: Popup with completion options

Why was it chosen to implement Since this is arguably one of the most helpful features in IDEs, the implementation thereof was paramount to the completion of this project. Programming in most environments, after the desired algorithm has been thought of, has become equal to the task of writing a few letters and then looking at the completion suggestions of the IDE.
Almost all modern IDEs offer a way to trigger code completion and this feature is heavily used. The main reason it was chosen to implement is that it brings very big improvements in productivity, freeing the programmer from having to remember all identifiers correctly and having to type them completely.

What benefits does it entail As already detailed, the improvement in productivity is huge when using this feature. A programmer that is used to code completion probably never wants to switch back to not using it. Amongst the existing solutions, only the Visual Studio integration offer code completion. This project however tried to go a little further and next to visually distinguishing the types of suggestions (e.g. methods, fields and so on), it also displays existing preconditions of methods in the suggestion popup.

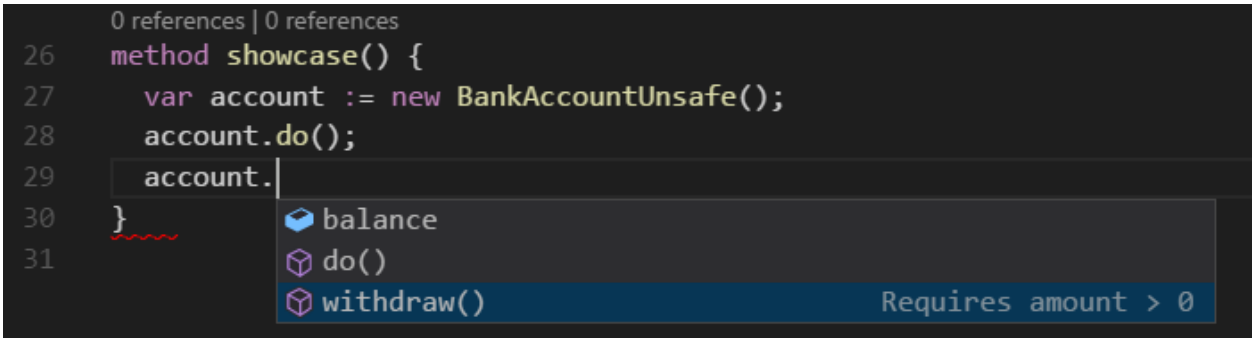

Figure 6.5: Suggestion displaying precondition

This combines a Dafny specific feature with a language agnostic one and lets the programmer know as early as possible under what restrictions he is working under. Being able to provide

code completion in this plugin provides programmers with the opportunity to translate their thoughts quickly into code and not having to bother with exact writing.

6.2.3 Go to Definition

What is Go to Definition Another common feature in modern IDEs is go to definition. It enables the programmer to quickly jump to the definition of a code element he is currently working with in order to gain further insight about it. This can usually be done either via a hot key for the current cursor position or an option when opening the context menu via a right click, Visual Studio Code offers both ways.

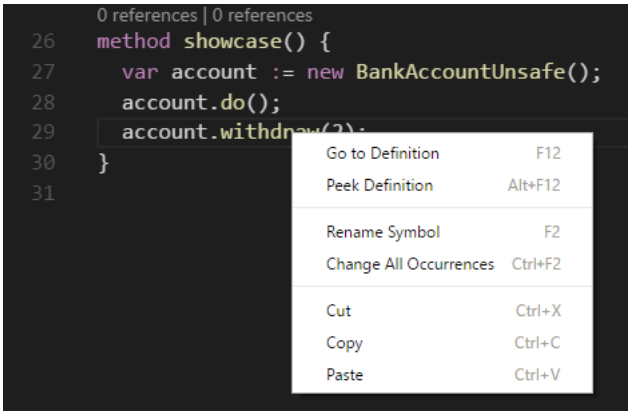

Figure 6.6: The Definition Features

Usually this feature offers to either open the code file containing the definition or just to peek at it in a popup. Visual Studio Code supports both these options. If the definition is ambiguous, which often happens when working with interfaces, usually a list of all possible definitions is displayed, because the concrete definition is often impossible to find using static code analysis.

Why was it chosen to implement As it is similar to codeLenses, this is a feature which is elemental to all modern IDEs and provides great overview over a project. The implementation of this feature had a high priority in this project. It is also used to navigate in a code base and is actually the reverse application of references in a codeLens, because instead of finding all usages, one jumps from a usage to the definition.
Next to code completion, it is probably the feature most widely implemented in IDEs, so it is important that a Dafny integration does implement it as well, as it provides programmers an idiomatic way of working with a code base.

What benefits does it entail Similar to codeLenses, go to definition is a command which can deepen the understanding of a code base profoundly. It is at all times clear where a certain symbol comes from, and one can easily check what it does by jumping to the implementation. It also helps a lot when one has to trace control flow backwards in order to find out where certain data originates from.

The existing solutions do not yet implement go to definition, with the exception of the Visual Studio integration. Being able to quickly navigate to a symbols definition keeps the programmer of having to keep track of all implementations and staying in the current scope of abstraction. If one needs to know more about a symbol, a peek can quickly show the desired information without distracting the train of thought with a new open file in the editor. This makes programmers that use this command usually more efficient and productive.

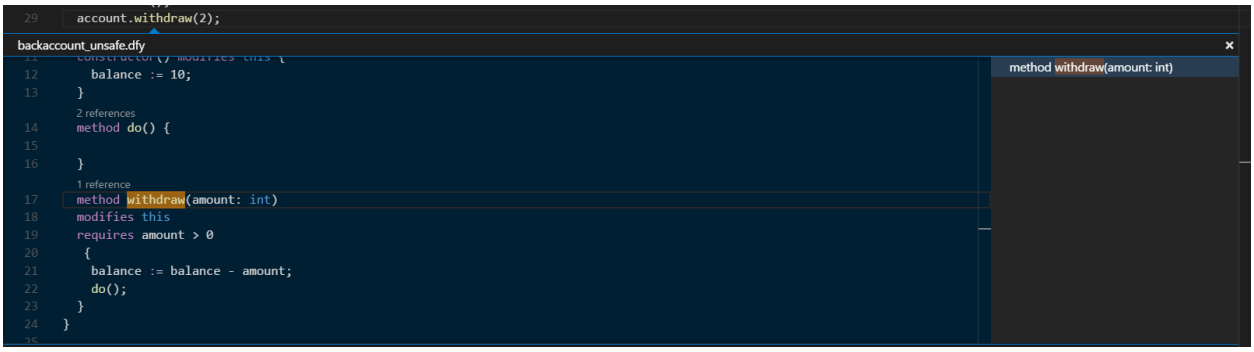

Figure 6.7: Overlay of peeked definition

6.2.4 Rename Element

What is Rename Element Rename element is a feature essential to refactoring. It is very widespread in IDEs. Visual Studio Code offers built in support for renaming either via a hot key or the context menu. Renaming an element switches all occurrences of an identifier to a new one. This can be a methods, an alias of an object or any other element in a code base.

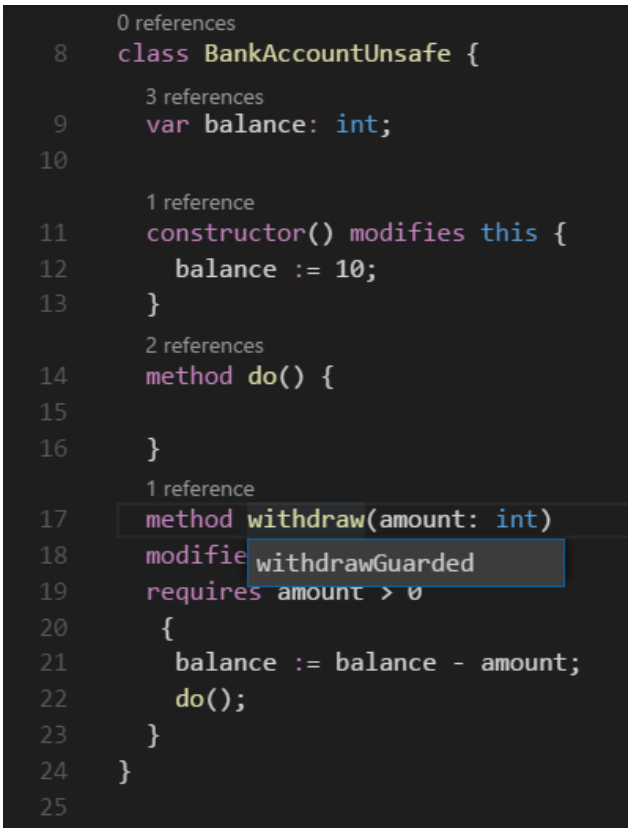

Figure 6.8: Renaming an element in Visual Studio Code

With this feature, one has to be careful when implementing it, since it actually changes the code base and it has to be sure which elements have to be renamed and which do not. There can be several identifiers that are exactly the same but reside in different scopes. When applying the refactoring, only the correct occurrences, which are determined by scope, should be changed while the others must stay untouched. Doing otherwise can change the expressed intent of a program or, if done poorly, can break code.

Why was it chosen to implement Because of its importance in refactoring, which is one of the most important tasks when programming, implementing this feature belongs to the core scope of this project. Renaming element allows to quickly make code better readable. Keeping a code base readable and clean should be one of the biggest concerns when developing a product, a feat which is only accomplished through constant refactoring, a task which is almost impossible to do when renaming of elements is not supported automatically should the code base grow sufficient big enough.

What benefits does it entail Renaming of elements allows the programmer to keep the expressed intent of a program in sync with the actual intent when implementations change as they tend to do over time. Normally more time in programming is spent editing or deleting old lines of code than writing new ones. Having automated support in this area does not only make work quicker, but also guarantees for non breaking changes when implemented

correctly. When one has to adapt many pieces of code manually, it is almost guaranteed to make a mistake somewhere.
None of the existing solutions support context aware renaming, which is very important if the refactoring should be applied correctly. When writing production quality software, renaming of elements helps to keep the intent clear and make the code base more readable, which is especially important if it has to be maintained for a long lifespan. For these reasons, a programmers that uses this feature often and correctly tends to write more expressive code.

6.2.5 Syntax Highlighting

What is Syntax Highlighting Syntax highlighting is the feature of displaying different elements in the editor with different colors. Usually there is an IDE idiomatic coloring scheme, so that for instance all language keywords or types are colored the same across different languages, if the IDE supports multiple.

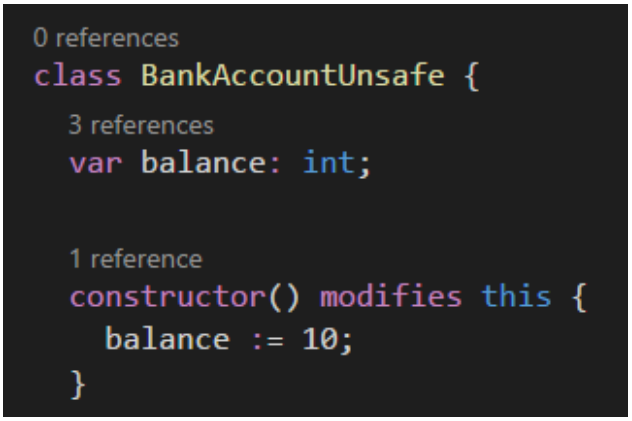

Figure 6.9: Syntax Highlighting of Dafny

Syntax highlighting is usually applied constantly and updates instantly when the programmer types new code. The mechanisms to display the coloring are also very deeply ingrained into the IDE, relaying on a declarative language grammar definition file to do all the coloring.

Why was it chosen to implement Syntax highlighting is definitely the first feature to get working in an IDE, maybe next to run program. Even simple text editors such as Notepad support syntax highlighting for multiple languages, so it is expected of a language integration to come with working highlighting.

What benefits does it entail This feature makes code a lot more readable, and a lot more time is spent reading code than writing. It makes the purpose of a symbol clear at first glance thanks to idiomatic coloring. Understanding code becomes much easier when one is guided by colors. Syntax highlighting can also help tracing bugs were the programmer and the compiler do not have the same opinion of a certain syntax, since syntax highlighting reflects the way a compiler interprets a piece of code.

Since this is such a basic feature, all existing solution support this feature, usually relying on the same grammar definition file written for the sublime integration of Dafny [tvi17]. This project expanded the implementation a little, also providing syntax highlighting for parameters and generic arguments, something that existing solutions neglect.

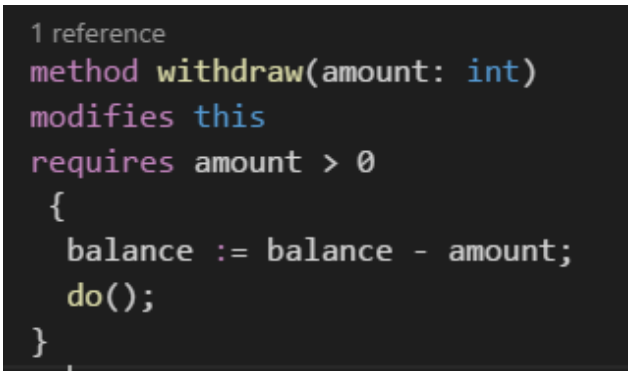

Figure 6.10: Syntax Highlighting of parameters

6.3 Dafny Specific Features

This chapter details the language specific features that were implemented during the project. Since Dafny focuses on an elegant way of writing specification constructs, the goal of these features is to aid the programmer writing them. While such constructs have great potential helping to write correct code, most programmers are not yet used to thinking in terms of this methodology. In addition, there are situations which appear often that warrant the writing of similar specification constructs. While important, this work can become tedious. The features detailed in this chapter try to mitigate both these points.

6.3.1 Counter Examples

What are Counter Examples Dafny's proof pipeline works by trying to find a proof by contradiction. This means, that when a programmer writes a specification construct such as a postcondition, Dafny and its pipeline try to find a situation where the postcondition does not hold. If no violation is found, the proof obligation is satisfied.
If however, a violation is found, that means that the pipeline has found a set of assignments to the variables involved in the context of the postcondition that led to the postcondition being violated. The set of such assignments that lead to a violation is called a counter example.
Counter examples are split up into different states. This means, Dafny includes all assignments to each variable in each state of control flow in a context. This may seem very abstract, but it becomes clear when shown in a simple example.

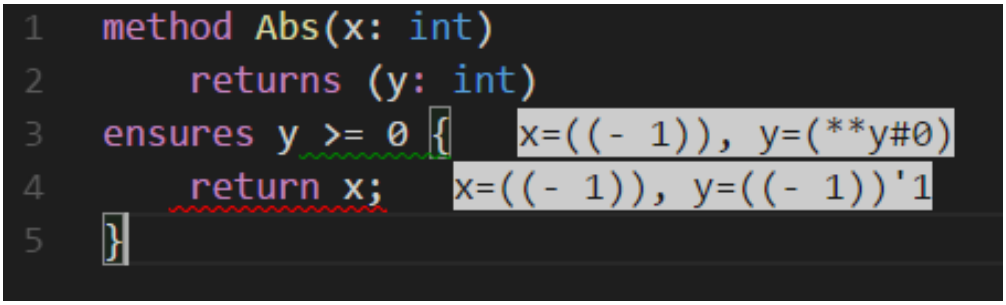

Figure 6.11: Counter Example is shown in Visual Studio Code

When a counter example is generated for a postcondition of a method as seen in 6.11, Dafny determines the value of each variable at each point of execution. In the example, there is only one line of code in the method body. This means that the method has two states. The first one is before the line is executed, in this case, only the parameter x is set to a value which will lead the postcondition to be violated. The return value y has not been set yet, so the counter model at this state does not yet hold a value for y.
At the next state of computation, the line in the method body has been evaluated. Dafny's counter model no not only shows the value of x, but also the value of y which can be determined after that line of code has been evaluated.
In summation, a counter model shows the values of all variables involved at all stages of computation that lead to a specification construct being violated.

Why where they chosen to implement To widen the circle of Dafny programmers, it is important to support Dafny specific features in a language integration. Displaying counter example was chosen mainly because of two reasons. The first one is that many programmers are not used to writing specification constructs, so displaying counter examples is a good way to clarify what a specification constructs actually expresses, deepening the understanding of the programmer.
The second consideration is that edge cases are a continuous source of errors. While they may often be forgotten when writing specification constructs, they are most likely the values that Dafny finds to build a counter model with. This quickly leads to the inclusion of edge cases when thinking of error free code, something which is difficult to achieve for IDEs.
A more internal notion on why they were integrated is that, as detailed in 8.2, the project failed to deliver a generic approach towards contract generation. Displaying counter examples is a way to partly make up for this missing feature, as it supports the programmer finding correct constructs.

What benefits do they entail Counter examples greatly deepen the understanding of code, as they bring to attention possible situations that the programmer did not think of. When writing complex code, it is nearly impossible to have all possible contexts in mind that it can run and possibly fail in. The displacement of counter examples helps the programmer reasoning on when implementations satisfy specification constructs, which is ultimately the main goal behind the design of Dafny.
They also help writing correct specification constructs, which is not always easy. When

constantly being shown counter examples on when a contract is violated, the programmer can refine a contract iteratively until it is no longer violated. This is also the approach that most modern proofing frameworks follow.

```
1  class BankAccountUnsafe {
     3 references
2    var balance: int;
3
     1 reference
4    method withdraw(amount: int)
5      modifies this
6      requires amount >= 0
7      requires balance >= 0
8      ensures balance >= 0 {      this=(T@U!val!48), this.balance=2275, amount=2276
9      balance := balance - amount;    this=(T@U!val!48), this.balance=((- 1)), amount=2276
10   }
11 }
```

Figure 6.12: Complex Counter Example with multiple specifications

As already detailed in the paragraph above, counter examples also often show edge cases that lead to a condition not being satisfied, a very important consideration when programming. These can then easily be taken care of appropriately by the programmer, either by changing the specification construct or the implementation. Without displaying this information, it is very difficult for the programmer to exactly find out how a condition is violated.
This project is the only solution that displays counter examples except for the Visual Studio integration, a feature which is probably the most important Dafny specific aid an IDE can provide.

6.3.2 Null Checks

What are Null Checks Null checks are specification constructs that take the form of a precondition that requires an object to be not null. They can be written whenever a new context is opened. Examples are when an object is given as an argument to a method, then the specification is a precondition to the whole method. Another one is when a object is declared, potentially not initialized and used in a later following while block. Then the precondition is written for the while block.

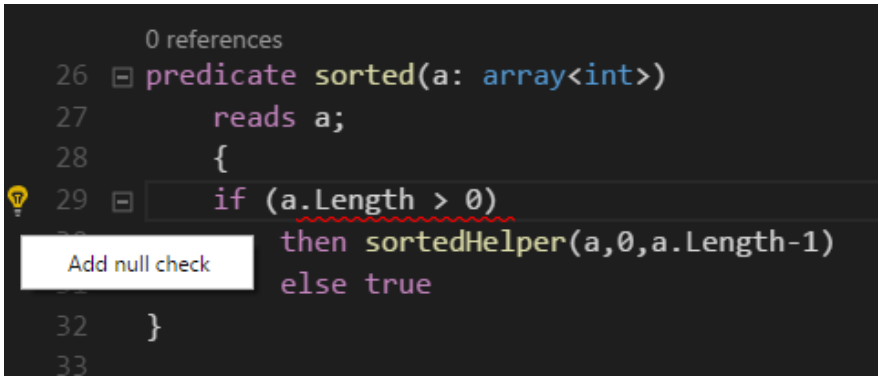

Figure 6.13: A situation that profits from a Null Check

When a null check is written in form of such a precondition, the following context can safely interact with the object, since it is sure that the object is initialized.

Why were they chosen to implement Programmers very often access members of elements, especially in the methodology of object oriented programming. While it offers a great way to structure a program and to represent reality in a program, it comes with some danger. The most common pitfall usually is the null reference. To help avoid this, Dafny reports potential null references. Since almost any piece of complex code deals with objects, this is a very common occurrence, since for instance every object given as an argument must be checked for null first.
It was therefore decided to offer a mitigation of this important, but tedious work. The insertion of such a precondition shifts the burden of providing valid arguments to the caller of a context, so the implementation can concentrate on providing functionality. While working with Dafny, it was noticed that such a precondition was needed for about every third method, and every other while block, meaning that a lot of work is done for the programmer by supplying this precondition generation.
Providing this Dafny specific feature was implemented as the first code fix, since a null reference arguably is the most common trivial error when programming in the object oriented methodology. Offering an automated solution to this in an IDE makes Dafny's strengths shine and working with it more enjoyable.

What benefits do they entail Current solution do not support the automatic generation of null checks, something which occurs very often when programming. Avoiding null references is important, but tedious work. Providing an automated fix for this enables the programmer to work in a safe context and to concentrate on providing functionality without having to think about checking the context first.

```
0 references
predicate sorted(a: array<int>)
    reads a;
 requires a != null {
    if (a.Length > 0)
        then sortedHelper(a,0,a.Length-1)
        else true
}
```

Figure 6.14: An example of a Null Check

This improves productivity, since it narrows the problems that the programmer has to deal with himself. Providing null checking therefore helps the programmer writing code more quickly and profit from Dafny in an automated way.

6.3.3 Bound Checks

What are Bound Checks When accessing an array, this is done by providing an expression that is used as an index into that array. Since an array has a fixed length, it is important that the expression resolves to a value inside the array's range.

```
59      {
60          while (a[i] < pivot)
        Add invariant
        decreases  pivot - a[i]    { i := i + 1; }
63          while (a[j] > pivot)
        invariant 0 <= j < a.Length decreases  a[j] - pivot    { j := j - 1; }
64          if (i <= j) {
65              var t := a[i]; a[i] := a[j]; a[j] := t;
66              i := i + 1;
67              j := j - 1;
68          }
69      }
```

Figure 6.15: Accessing an array without making sure the index is in bound

Since the definition of an array's range is always the set of integers between zero and the array's length minus one, bound checks can be implemented as two specification constructs that take the form of preconditions belonging to the context in which the array is accessed in.
The first precondition states that the expression used as an index must resolve to a value bigger or equal to zero, the second one that the expression must resolve to a value smaller than the array's length. When these two preconditions hold, it is sure that the array access will succeed and not result into a memory violation.

Why where they chosen to implement Since Dafny has enough knowledge about the array data structure, it issues compilation errors every time an array is accessed without checking for its bounds first. While programming with Dafny, it was observed that this is the case in almost every non trivial example using arrays. Since the array data structure is used quite often in Dafny, it was decided to also help the programmer with this construct, since bound checking is important, but tedious.

What benefits do they entail Current solution do not support the automatic generation of bound checking. This task has to be performed very often and is tedious. Relieving the programmer of this repetitive work frees his mind for working on core functionality of a product.

```
52  method FindMiddle(a: array<int>, start: nat, end: nat)
53      returns (middle: int)
54      requires a != null
55
56      requires  0 <= (start + end) / 2 < a.Length
57      requires (start + end) / 2 < a.Length
58
59  {
60      middle := a[(start + end) / 2];
61      return middle;
62  }
```

Figure 6.16: Bound checks were generated

Automatically resolving these checks therefore makes working with Dafny more efficient and is part of making the plugin more usable and supportive.

6.3.4 Increase / Decrease / Invariant Guards

What are they These constructs that are called guards in this paper help ensuring that an expression complies with certain conditions. This is done by writing specification constructs that ensure this.
In case of increase or decrease guards, this is done by either writing an increase or a decrease specification construct, demanding that an expression either increases or decreases. This is often needed to meet loop termination or when using recursion, in both cases Dafny recognizes that an expression should change over time in a certain way and requires constructs detailing this. This makes sure that for instance recursion reaches a termination clause or a loop terminates.

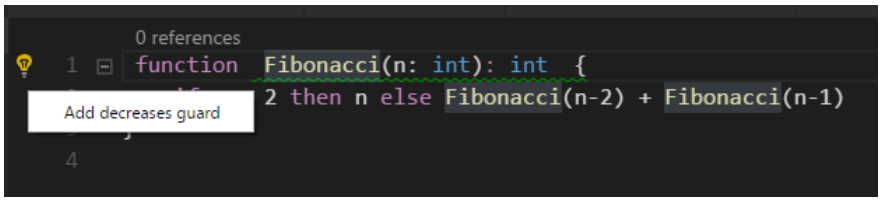

Figure 6.17: Recursion should always decrease an expression

In case of invariants, it is done by writing a specification construct called an invariant. When an expression used in an invariant can be dynamic, it specifies to which range of values it can resolve. This is for instance needed when an expression used as an index to an array is dynamic inside of a loop. The invariant then makes sure that the expression is always in bound of the array.

Why were they chosen to implement The first two guards are needed to make sure an expression converges to a certain range of values over time. This is the case when making use of recursion to ensure that the base case is eventually met, or when writing loops that depend on a certain value of an expression for termination. Both examples are very important, because not handling them correctly can result in endless loops or overflow. Dafny already does a good job in generating warnings that tell the programmer that constraints should be enforced for a certain expression. These situations also occur very often, since both recursion and loops are fundamental elements of programming.
Often, an expression must always evaluate into a certain range, this is for instance the case when an expression that is used as an index to an array is not constant within the context of a loop.
Since the above detailed constructs are used often, and the appropriate specification constraints help using them correctly, it was decided to implement an automatic way of generating them.

What benefits do they entail No existing solution offers an automatic way of generating these guards, a tasks which soon was discovered was tedious and time consuming, because it is not always clear at first glance which expressions are subject to constraints. Providing an automatic way of generating them does ensure that the specification constructs are declared correctly, manually inserting them is error prone and demands some reasoning by the programmer.

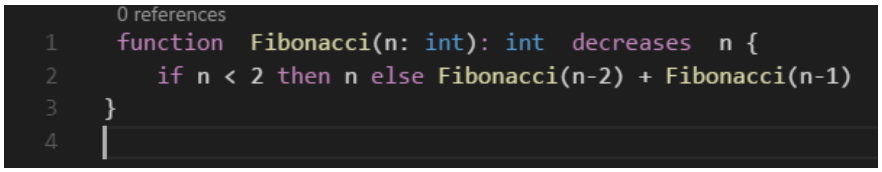

Figure 6.18: Recursion guided by a decrease guard

This feature therefore allows the programmer to be more productive and concentrate on key areas of concern.

6.3.5 Flow Graphs

What are they A existing feature in Boogie is to translate boogie programs into a flow graph. It does not only show the flow of the program itself but also includes all pre- and postconditions. Below is an example of a flowgraph, taken from the partition method of a quicksort algorithm.

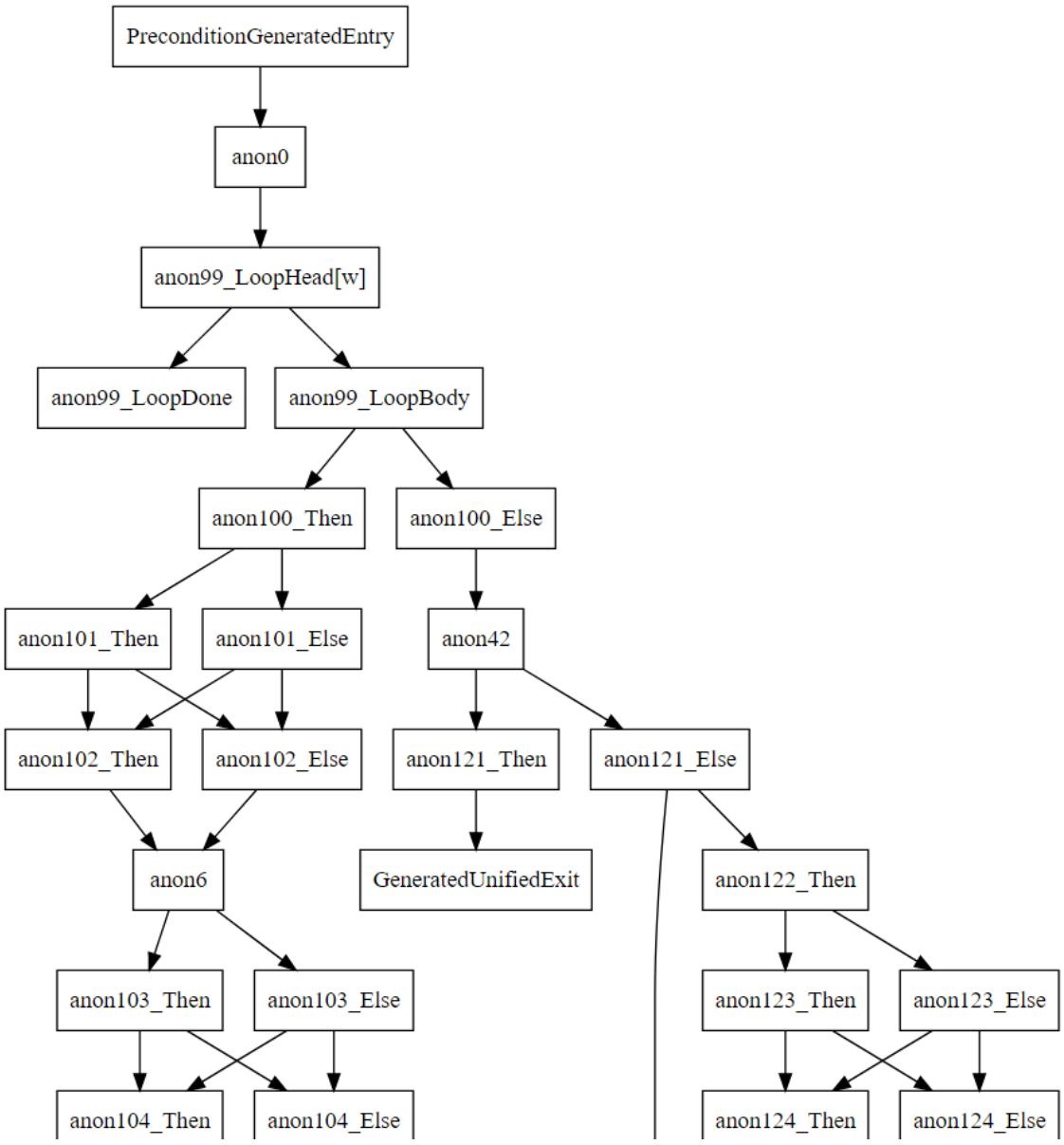

Figure 6.19: Flow Graph of the partition method

Why were they chosen to implement Because this feature existed already in Boogie, and the goal was to support as many features from the Dafny pipeline as possible, it was decided that this feature could be easily and nicely implemented.

What benefits do they entail Developers can directly see the flow of the program they are working on in Visual Studio Code. Otherwise, they would need to manually save the graph to a file, upload or convert the file to finally view it. This all is done in the background to provide a nice user experience.

7 Possible points for Extension

This chapter details what further work could be done on the plugin by subsequent projects.

7.1 Support for other IDEs

Since the plugin is structured as a language server, it should theoretically be possible to integrate it into any IDE which implements the language server protocol without any problems. In reality, some customization is often needed and some popular IDEs also do not fully implement the protocol yet. This chapter details which IDEs were looked at as possible hosts for the plugin and what would have to be done for a complete integration.

7.1.1 Eclipse integration

The eclipse project LSP4E aims to integrate existing language servers into the Eclipse IDE in an easy way.
"It includes some APIs to turn language server protocol elements into Eclipse IDE concepts and a generic integration allowing to easily plug any language server to an Eclipse IDE instance without need to write Java code, either via a plugin associating a new language server, or by letting users manually bind language servers to their IDE." [Fou17] It is built on top of LSP4J, a Java implementation of the language server protocol.
Integrating the Visual Studio Code Dafny language server into Eclipse could become possible in the future. Right know there is no way to interact on the client side. One only can specify a language server based on a program (NodeJS) and arguments (extension.js) which is executed and used for querying information. One cannot add any behavior to Eclipse itself, which would be necessary for certain features. Also, the sendRequest protocol specification is not implemented yet (Commit: 1615e07), which is important for starting the DafnyServer correctly. The better way would be to program a new Eclipse plugin, based on LSP4J, which then would also allow to customize the status bar, run scripts and show progress information.

7.1.2 Emacs integration

Emacs [GNU17] is a versatile IDE which enjoys great popularity especially in the open source community. Written in LISP, Emacs traditionally supported language integration via so called modes, which are demons that run in the background. This is already a similar architecture as used in a language server integration.
Work on integrating language servers into Emacs has already be done. The project emacs-lsp [ema17] aims to provide the connection between Emacs and language server. The project itself is structured as a classical Emacs demon and allows interactions with a language server. There already exist some integrable language servers in languages such as Java or Haskell. The integration of emacs-lsp and existing language servers seems to be pretty trivial, an example can be seen in [Ema17] using Java.
Since the language server allows custom messages to be defined as detailed in the implementation documentation, the Dafny plugin defines some of them. One set of custom messages is necessary, since the plugin allows for downloading the Dafny compiler automatically. The

communication in regard to this feature must be extended to the IDE, so that the user is aware of this feature. Additional custom components are the state of the file so that the programmer sees if it is verified or not, or the displacement of counter examples for proofs. These features were deemed very useful in this project, such that they should be a part of every IDE integration. This means that a little wrapper would have to be written on the client side of the language server, which understands the custom messages and can relay this information to the IDE itself. The work that has to be done is trivial, since it simply means to pass information on and display them accordingly.
It was decided that gaining a working knowledge of LISP and Emacs in order to write such a wrapper was not in the scope of this project. For a programmer familiar with Emacs and LISP this task should take no more than a week. The custom messages that need to be implemented can be found in the implementation documentation. It can be argued that this extension should be one of the first ones to be tackled by later work, as one can gain a considerably larger user base without having to invest too much work.

7.1.3 Monaco integration

Monaco[Mic17b] is the code editor that powers Visual Studio Code. It is also possible to use Monaco as a standalone editor in the browser. It would be intriguing to integrate Dafny directly into Monaco, since there are not many simpler setups imaginable as opening a browser window. It was therefore decided to look into a possible integration of the plugin during the course of this project.
It soon became apparent that the project does not seem very active, as the latest commit was two months old at the time of this writing. The documentation for developers is also very sparse. There was hope that an integration would be trivial, since the plugin is structured as a language server. However, Monaco does not implement the language server protocol. In the documentation, it can even be read that "Extensions written for Visual Studio Code will not run in Monaco"[Mic17b], without any further explanation given.
However, integrating a new language into Monaco is possible, as can be seen in an example for Typescript in [Mic17d]. However, the setup is different than a language server integration or a Visual Studio Code plugin. It is questionable how much code could be shared. While the idea of running Dafny in an editor in the browser is interesting, the work that would have to be done in order to achieve this was deemed to be outside of the scope of this project. In addition, if further work aims to implement this integration, an evaluation on how active the work on Monaco is should be done first, given the apparent standstill in the project at the time of this writing.

7.2 New Features

This section details some interesting ideas that were gathered during development of the plugin, but were sadly out of scope of this project. It is thought as a starting point for developers that want to extend this plugin.

7.2.1 Debugger

Integrating a debugger into the plugin would be great, since this is a feature that is very often used and very helpful when searching for bugs. An existing solution already supports this feature, namely the Visual Studio integration [Mic17e] of Dafny. This already works very well and can be used as a guideline when developing an own integration. It is also a feature that programmers have become used to in all modern IDEs, so it should be part of any language integration.
Sadly it is also quite a difficult task since the interaction between an IDE, an executable and a debugger is complex. While the Visual Studio integration is open source, it is written in C# and can therefore interface directly with the Dafny pipeline, which is heavily done in that integration. It would take quite some work to abstract this direct interaction into a clean API that extends the existing API of the Dafny pipeline. Using this API, a language server could provide an own implementation of a debugger.
Designing this complex component was well out of scope for this project. It is estimated that, depending on the preexisting knowledge, this would be an endeavor of about two to four weeks time and probably would have to rely on at least some help by the Dafny development team. While this task is difficult, the result would be having an important feature that all users of modern IDEs anticipate in a language integration. It therefore should be a primary consideration when deciding on how to extend this project.

7.2.2 Widening Scope

While many features have been implemented during the course of this project, the scope of their application was often narrowed as to provide a pragmatic approach to the problems regarding the scope of the project. For example, the rename element refactoring only works on members of classes, but not on parameters of methods.
The exact scope of all features can be found in the implementation documentation of this project. While the extension of scope of existing features might seem tedious and not very interesting, it can introduce subtle problems that have to be dealt with with great care. It also is beneficial to the users, as they can apply features in much wider contexts.

7.2.3 Contract Generation

While contract generation has been implemented for some often occurring situations as detailed in 6.3, there is still a big potential for improvement in this area. Since a generic approach to contract generation has been deemed to be unfeasible, the next best way is to offer help in writing specification constructs in specific situations.
When deciding to do work in this area, it is important to first analyze which situations appear often in a typical Dafny program. While this task it is difficult in itself, since it requires some expertise in Dafny, it is important because otherwise features may be implemented that do not get used in every day scenarios.
When such situations have been identified, the specification construct generation must absolutely be correct. Since this is a feature that changes existing code, it is better to not provide any help when there is ambiguity regarding the correct solution. However, when

done correctly, this is a feature from which programmers can greatly benefit, since it enhances productivity by eliminating the need to do complicated reasoning. It also introduces capabilities of Dafny to programmers that are not yet proficient in their usage of the language.

8 Conclusion

The section concludes this paper by first comparing the reached goals with the stand of the existing solutions. In the end, an evaluation of the project is detailed. Further insight regarding the implementation details of the project can be found in the implementation documentation, which accompanies this paper.

8.1 Goals Reached

This chapter details the goals that were reached in this project and puts them into contrast with the current solutions that were introduced in 4.2.

8.1.1 Platform Independence

The chapter 6.1 details how the plugin runs on all major platforms. While only one existing solution, namely the one offered for Emacs, runs on all platforms, the user still has to make sure to have the Dafny pipeline configured correctly for his environment.
This project has developed the only solution which is able to run the whole pipeline on all operating systems and does so automatically. This contributes in reaching the biggest user base possible and invites all programmers to try out Dafny without having to switch the environment that they have become accosted to.

8.1.2 Setup

In this area, the product developed during the course of this project surpasses all existing solutions. While those all require the user to install and configure the Dafny platform, this plugin handles the whole setup automatically. The way this is done is detailed in 6.1. The Visual Studio Code plugin can be installed by one click within the marketplace that Microsoft offers, while the whole dependency resolution resides in the language server part of the plugin, meaning this automatic installation is also possible when integrating with other IDEs.
Having unit tests in place which hinder breaking changes and doing all the resolution in a central place lays the foundation of a setup that is as simple as can be. This invites programmers to try out Dafny without having to invest any energy and is an important part in widening the user base.

8.1.3 Usability

The language server protocol details both standard and custom messages than can be emitted by the server. This was used during the work on this project to relay almost all information emitted from Dafny to the programmer. The plugin also translated this messages in idiomatic Visual Studio Code notifications, giving them the appropriate form or displaying them in the correct GUI element. It can therefore be said that this project leverages all elements that Visual Studio Code offers and programmers familiar with the most common GUI driven IDEs should have no problem working all the features that were implemented.
While already all existing solutions integrate nicely into their respective IDEs, only the

Visual Studio integration details the same amount of information as the plugin developed in this project. This makes it attractive for a wide range of programmers that are used to working with different IDEs.

8.1.4 IDE Independence

While developing this plugin, it was decided to implement it as a language server. This has the potential to decouple the plugin from a concrete IDE and make it reusable, as was detailed in 6.1. Making such an integration to an IDE other than Visual Studio Code work was outside of the scope of this project, but a proof of concept was done for Emacs.
The proof of concept and the fact that the protocol is gaining traction fast make it safe to say that this project hat laid the foundation for many potential IDE integrations to come. 7.1 gives some ideas which integrations would be feasible in the near future.
The approach chosen when designing this plugin as a language server is unique among the existing solutions, making it the only one that is not hardwired to a specific IDE. This has great potential of widening the Dafny audience, but also of inviting other programmer to contribute to the plugin.

8.1.5 Feature Richness

During this project, many different feature were implemented. This includes language agnostic features detailed in 6.2 as well as features that lend Dafny specific support to the programmer detailed in 6.3. While it was an important concern that programmers can use most features that they are used to from working with other languages, much thought was spent on how programmers can be supported writing Dafny specific code. This resulted in a collection of some solutions that are unique to this project.
Comparing the stand of the project with existing solutions, it surpasses most vastly regarding feature richness in both areas. The only platform to offer features that this project does not do, is the Visual Studio integration. These features include a Dafny debugger, which is very helpful but sadly was outside of the scope of this project. However, this project integrates some features that have not been implemented yet in any IDE, especially automatic specification construct generation for some situations.
Since most solutions are open source, it stands to hope that they continuously offer more features by getting inspiration from other solutions. At the current time, programmers that make use of the solution developed in this project benefit from one of the two most feature rich Dafny programming experiences.

8.2 Evaluation

Following the comparison made in 8.1, it can safely be said that product developed during this project is at least en pair with most existing solutions. An automated setup of the entire environment which resolves dependencies on all major platforms guarantees that new users can make their first steps with Dafny without having to spend time to get things running. This is an important first stepping stone in making Dafny accessible for a wide user base.

After having gotten Dafny to run on an environment, the next concern is to offer a productive, efficient and enjoyable programming experience. This keeps programmers using Dafny interested and keen on further using it. Having implemented many common IDE features as well as some Dafny specific functionality lets programmers apply work flows they have established with other languages as well as discovering the big benefits regarding functional correctness Dafny can offer.
Structuring the plugin as a language server decouples it from any particular IDE, although this project relied on Visual Studio Code as a reference IDE. This decoupling leads to the possibility to integrate Dafny into many other IDEs without having to duplicate any logic. This further lays the groundwork to support a wide user base as a programmer would not have to switch the IDE he has come used to for another one.
Offering generic contract generation sadly was established to be not feasible to implement in the scope of this project. During this research it was found that this problem probably also is unsolvable due to the proof pipeline that Dafny uses. Instead of offering generic generation, often arising situations were identified and aids provided for them.
It was a major concern for this project to not just develop a proof of concept, but a production ready release of the plugin. Through the application of continuous integration feedback from users could be integrated early on. The plugin has been already downloaded over 300 times from all over the world at the time of this writing. It was also used in a course concentrating on software engineering at a school for higher education to get students familiar with contract based reasoning. During this, the plugin was used by about 60 students, underlining the robustness and platform independence of the solution. The minor changes that had to be made to the Dafny pipeline itself during this project have been accepted by Microsoft and are integrated into the official version of the pipeline.
The open source nature of this project encourages people to participate in enriching the plugin, as well as integrating it in additional IDEs. Close detail has been paid on keeping the documentation as well as the code quality at a high standard, so contributing should be possible without having to invest too much time to get started. It stands to hope that providing the plugin in the state it is at the end of this project is a big stepping stone in widening the user base that can profit from using Dafny.

A Project management

This section details how the work was organized. This entails the planning of the work as well as at how the workload was split up. In addition, the measures taken to ensure the quality and progress of the project are also listed.

A.1 Project Plan

This project plan was the result of the original planning of the project. It put milestones, features and artifacts into a desired chronological order. Some deviations of this plan had to be done during the project because of new insight was gained while working on implementing the plugin. These deviations are detailed in A.4.

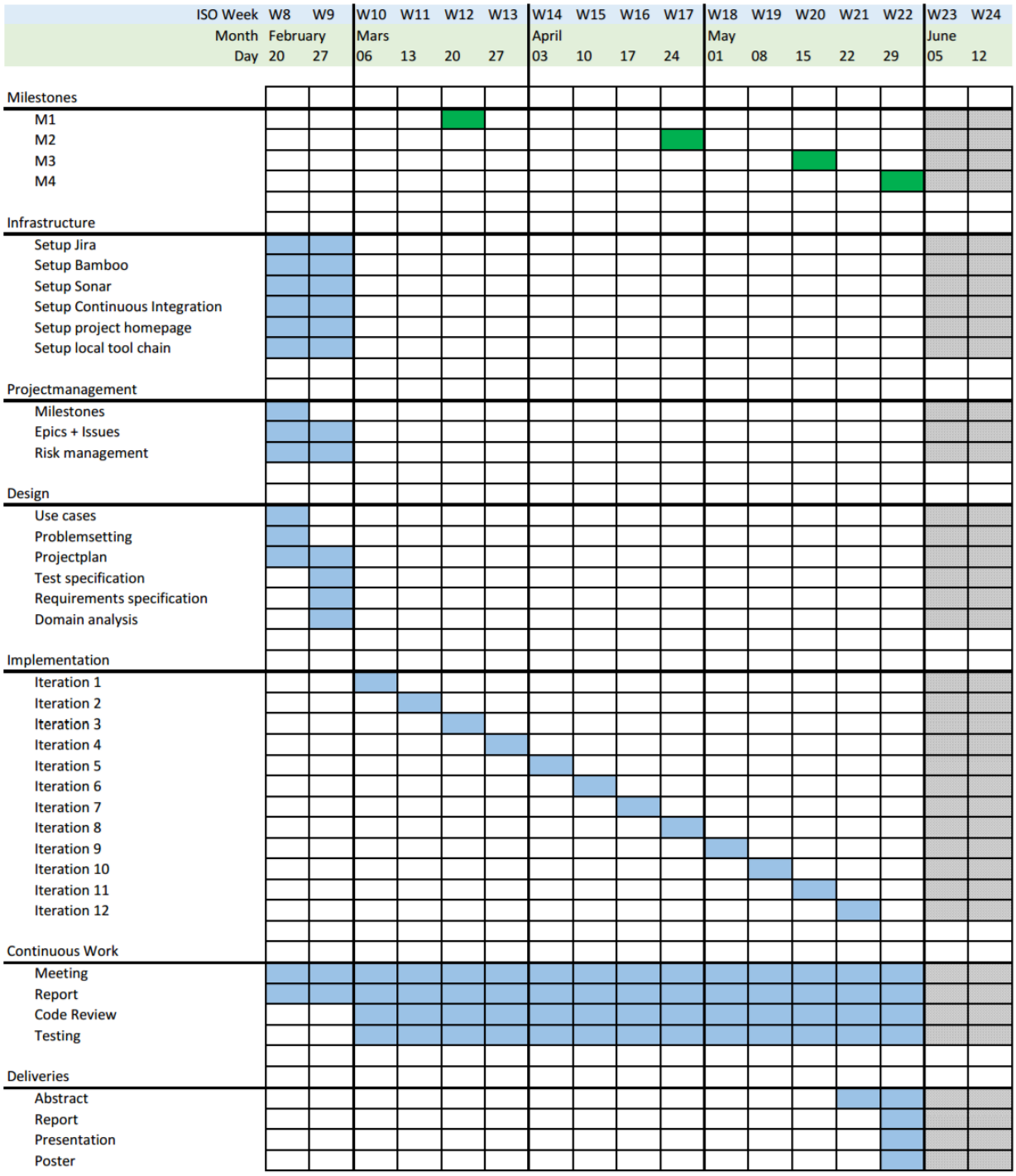

Figure A.1: Project Plan

A.2 Milestones

This chapter defines the milestones that are referenced in A.1. As with the project plan itself, some milestones could not be reached as they were defined, because insight was gained while working on the project that made implementing these requirements unfeasible without adjusting them. These deviations are detailed in A.4.

Nr	Date	Title	Description
M1	26.03.2017	First prototype	Plugin is working on all operation systems, simple verification is in place and errors are reported, basic syntax highlighting, automatic downloading of dependencies and installation of Dafny and configure of environment variables
M2	30.04.2017	Contract generation	First automatic contract generation, customizable with DSL
M3	21.05.2017	IDE features	Auto completion for identifiers, adding support for common refactorings like renaming
M4	04.06.2017	Release 1.0	Everything implemented and tested. Ready to be published

Table A.1: Milestone

A.3 Risk management

This section details what risks were identified in the beginning of the project and the potential impact they could have on reaching the main goal of this project. Next to the risks, mitigations are also defined. In the end, the risks are put into relation in a risk matrix. This helped in deciding which risks can be neglected and which ones must be mitigated at the very start of the project.

A.3.1 The Risks

Nr	Title	Description	Max Harm [h]	Probability	Prevention	Behavior at entry
R1	Underestimation of workload	One or more team member cannot finish a feature in time or loses too much time on a minor feature	70	25%	Weekly scrum meetings with feedback about the current work progress. Estimate workload together. Include time reserve	Help each other, if one is struggling. Feature reduction as last resort.
R2	Work flow is not working	Tool chain is not working as planned. The used components are not ideal for the problem	30	15%	Use experience to get a good setup and test it a lot in the beginning	Redefine tool chain or change single tool
R3	Communication problems	Team is not working together and each member is developing individually. Creation of incompatible interfaces. Talking with the industrial partner is not possible as expected, due to time difference or not enough time	70	20%	Weekly scrum meetings and agreements. Already worked in other projects as a team before	Increase written documentation and working closer together. Having a fixed meeting with the industrial partner

R4	Lack of knowledge	Team has too little knowledge about Dafny, Visual Studio Code and the development of a Visual Studio Code Plugin	80	10%	Already gain experience with typescript for plugin development, as preparatory work informed about Dafny	Find necessary knowledge online, get help from Microsoft Research
R5	Data loss or manipulation	Data loss because of a server crash or open permissions to access and modify data	20	3%	Regular backups and restrictive permissions to change files	Use backup to restore data. Extend safety measures
R6	Failure developer machine	One of the developer machines is not working anymore.	10	1%	Limited, regularly commits and backups.	Switch temporary to the school computer and install the necessary tools.
R7	Implementing the DSL is too complicated or too difficult to use	The DSL would be needed to suggest possible contracts or to customize the best practices.	100	30%	This core feature has to be researched carefully so that it really fits.	Reading about DSL, talking to people which have already worked with it.

R8	Supporting all features on the different environments is not possible	Due to that Windows, OSX and Linux are quite different, it could be that some features are not possible to implement on a platform or cause a big overhead to get it running	10	25%	Using as much of the standard API as possible. Implement specific features platform dependent	Disable a certain feature on a platform or look for a workaround.
R9	Automatic installer of Dafny is not possible	Maybe because of security restrictions, it could be possible that you cannot download and install additional software and set environment variables.	35	20%	Test if downloading of software can done inside VS Code.	Automatic installer cannot be done inside the plugin. Switch to a different strategy and implement an automatic installer over an additional executable, which could also install the plugin in Visual Studio Code.

R10	Dafny and the way it is used is not well understood	If it is not clear how exactly the work flow of a Dafny programmer could be enhanced, there is the possibility that a lot of features could be implemented that do not improve the programming experience at all.	80	20%	Regular feedback loop with people that use Dafny.	Learn a lot about Dafny ourselves, talk to people that use Dafny.

Table A.2: Risk management

A.3.2 Risk Matrix

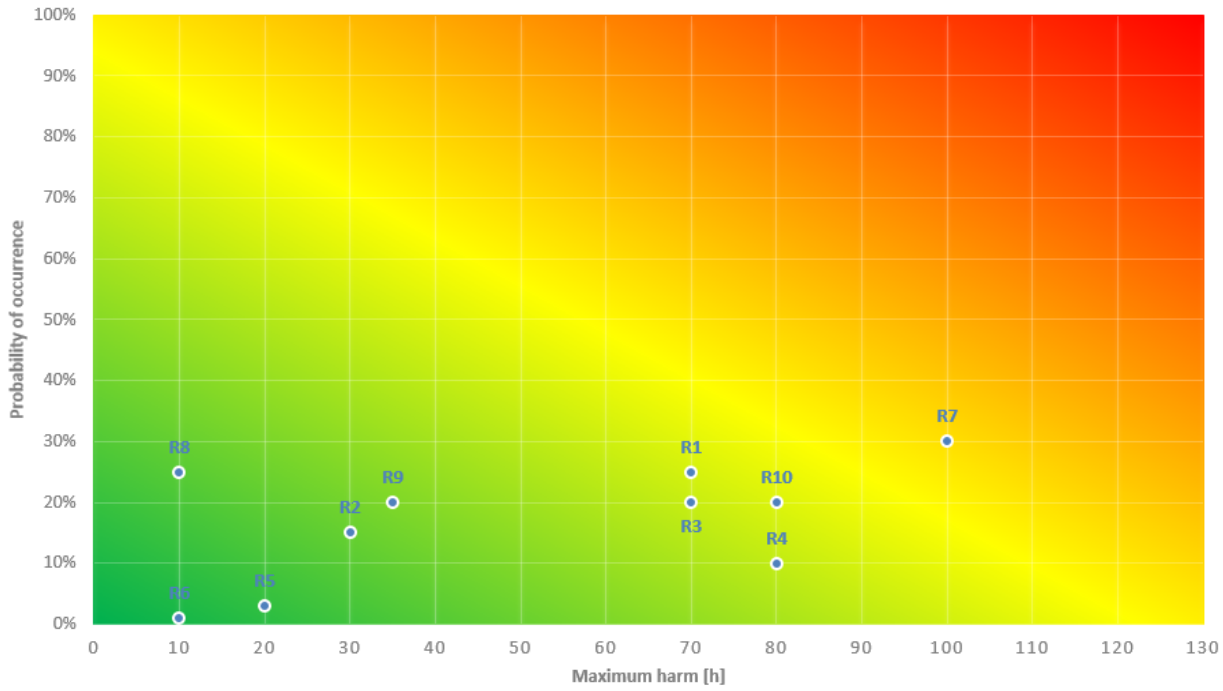

Figure A.2: Risk Matrix

A.4 Deviations from the project plan

This chapter details how the project was implemented and in what way it deviated of the project plan.

A.4.1 UC3: Reporting of Dafny best practices violations

While this idea seemed obvious when planning the project, this feature could sadly not be implemented. When working with different IDEs and well established programming languages, a programmer is often used to be supported by tools which can help write cleaner and more idiomatic code, such as linters. From this viewpoint, the integration of such a tool into the plugin seemed necessary.
While established languages have a pool of agreed upon best practices, Dafny is still a young language with not wide spread usage yet. The tooling around Dafny is also still not as sophisticated yet as for other languages. From this it can be inferred that there is not a big collection of programs yet to gain experience from, and that the problem of establishing a clear work flow with Dafny usually still exists, further preventing programmers to concentrate on idioms.
These facts reflect why there is no collection of best practices for Dafny yet, either in form of some documentation or as a suggestion from people that are involved with Dafny.
A second point worth noting in regard to this use case is how best practices for Dafny could look like. Dafny differs to most other languages in that it provides excellent specification

constructs. A natural area to agree on idioms would therefore be the contracts of a piece of code. This could also greatly enhance the performance, because if clever usage is made of techniques such as short circuiting, a proof can be calculated at a much lesser cost than with a naive implementation.
While providing support for best practices for contracts would therefore be very nice, this would also be almost unsolvable complex. Even for simple cases a deep understanding of proof theory would be needed, while for complex conditions it is not determinable if they can be proven in a better way or if they can be specified and proven at all.
Since the establishment of own best practices for Dafny without being able to include an existing collection was deemed an unrealistic goal, and the structuring of contracts in the best way possible an unsolvable task, it was decided to concentrate on other features of the plugin instead.

A.4.2 UC4: Automatic generation of contracts

Since the biggest selling point of Dafny is the possibility to write specification constructs. It therefore was clear to try to provide automatic generation of some of such constructs in this project. While planning the project, the proof pipeline used by Dafny was not yet understood fully, the grasp on proof theory was quite small as well. This made it very hard to estimate if such a feature could be implemented at all and if so, in what time. Nevertheless the potential benefit of such a feature marked it as a milestone in this project.
While researching the theoretical basis for implementing such a feature as detailed in 5.1 and the chapters following it, it became apparent that the topic was quite complex. A first stumbling block were invariants. As languages such as Eiffel [Mey17] make it possible to work with invariants, it was assumed that Dafny offers this possibility as well.
As was learned, there are several different methodologies when it comes to invariants, having them implemented as a macro which simply inserts them as postconditions to every block is the easiest approach. A collection of techniques can be found in [Dro+08]. Dafny does not build in object invariants because it does not commit to a particular methodology. Since the upholding of certain business rules (which was the aim of this use case) via specification constrains usually translates into generation of invariants, in addition to generating the invariants, it would have also been in the scope of this project to define how to deal with invariants in Dafny in a consistent work. It was also impossible to define and implement a way in which such business rules could be expressed regarding time and the trade off between usability and flexibility.
The second idea was to apply the concept of the weakest precondition. This means that if a proof does not hold given a context, one can find the weakest precondition to make the proof valid. It would have been great to offer the generation of the weakest precondition as a refactoring in the plugin. However, since the weakest precondition must be expressed in Dafny, it would have to be found in almost human readable form. Z3 [Z317], the theorem prover used by Dafny, tries to prove a theorem via contradiction. It is very hard to gain information about satisfiability from the prover. In addition, there is the problem that Dafny code gets translated to the Boogie[Mic17a] meta language, which then gets translated to Z3 syntax. Even if one could gain information from Z3 about satisfiability, it would therefore be difficult to provide a matching back to the Dafny language and to display this information.

In general, the problem of inferring sufficient conditions for proofs is very hard if they should be human readable. The few existing solutions work with an iterative approach relying on stepwise reduction of counter examples, for instance described in [SK13]. This approach is difficult to implement and lacks the usability that was sought after in this use case.
Lastly, it took a lot of time to gain a working knowledge of the proof pipeline. Since it consists of three big projects, namely Dafny, Boogie [Mic17a] and Z3 working together, a deep understanding was not possible without having some existing knowledge in this time. While trying to understand the pipeline, it also became apparent that the knowledge of the authors in proof theory was not deep enough to really grasp the problem and work on a solution in a feasible time.
Because of all these reasons, it was decided to not implement this use case in this project. While this is regrettable, other ideas were gained during the investigation of the problem. The feasibility of displaying counter modules was discovered, as well as small refactorings that provide constraints for a small, but very often needed set of instructions such as array accesses were thought of. The remaining time of this milestone therefore was directed at implementing these features instead.

A.4.3 Code Actions

As detailed in A.4.2, while trying to implement a proof of concept for contract generation, some often used concepts while working with contracts were discovered. This led to the idea to offer refactorings to generate the correct contracts for these concepts. While this is in no way a generic approach towards contract generation, it seemed as though these refactorings could help in many situations, making life considerably easier for the programmer. It was therefore decided to implement them, partly replacing the goals sought after in A.4.2. A more complete picture, including an overview of the implementation, can be found in the implementation documentation. The following list details on why these concepts were chosen.

Null Check Programmers very often access members of elements, especially in the methodology of object oriented programming. While it offers a great way to structure a program and to represent reality in a program, it comes with some danger. The most common pitfall is what Hoare famously declared his biggest mistake [Hoa09], the null reference. To help avoid this, Dafny reports potential null references. Since almost any piece of complex code deals with objects, this is a very common occurrence, since for instance every object given as an argument must be checked for null first.
It was decided to offer a mitigation of this important, but tedious work. The plugin detects warnings about potential null references, and offers to generate a precondition that demands that the designator standing for the potential null reference may not be null. If the programmer accepts the proposal, the precondition is inserted at the correct location. This shifts the burden of providing a valid context to the caller of the method, so the method can concentrate on offering a solution to the call. While working with Dafny, it was noticed that such a precondition was needed for about every third method, meaning that a lot of work is done for the programmer by supplying this precondition generation.

Bound Checking Almost as often as checking for null, it is necessary to check if an index to an array is in bound. Dafny already has sufficient knowledge about the array data structure that it issues a warning every time it is not clear if an index that is used to access an element is in bound of the array. While programming with Dafny, it was observed that this is the case in almost every non trivial example using arrays. Since the array data structure is used quite often in Dafny, it was decided to also help the programmer with this construct, since bound checking is important, but tedious.
Whenever a warning is issued by Dafny that an index may be out of bound, the plugin offers to generate two preconditions, namely one that states that the expression representing the index must be bigger than zero, and one that states it must be smaller then the array length minus one. The placement off course must be so that the preconditions are introduced at a place in the program, where all variables used in the expression have been declared. When the programmer decides to use the quick fix, the preconditions are inserted and relieve him of the hassle of manually checking the bounds of the index.

Increase / Decrease / Invariant Guards Another concept that often arises when using Dafny is to make sure an expression converges to a certain range of values over time. This is the case when making use of recursion to ensure that the base case is eventually met, or when writing loops that depend on a certain value of an expression for termination. Both examples are very important, because not handling them correctly can result in endless loops or overflow. Dafny already does a good job in generating warnings that tell the programmer that constraints should be enforced for a certain expression.
These situations also occur very often, since both recursion and loops are fundamental elements of programming. It was decided that the generation of these constraints could greatly benefit the programmer. In order to do this, the plugin offers to add an increase / decrease clause with the correct expression in place when Dafny detects recursion or a loop. When the programmer chooses to use the quick fix, the guard statements are inserted at the correct place. Another important constraint when working with loops goes hand in hand with the use case described above, namely when an array is accessed within a loop and the expression used as an index is not constant within the context of the loop. For this, Dafny offers the construct of invariants, that ensure that an expression is within a certain range during a given context. The plugin therefore offers to generate invariants, which ensure that an expression used as an index is always in bound of the array. When the programmer chooses to use this feature, the invariant is inserted at the correct context.

A.4.4 Counter Examples

A concept which was not known as the project started were counter examples. They provide a huge benefit as to understand how a program violates its contract. Especially if it is a more complex software with many methods and many branches, it can be very difficult to understand why a contract does not hold
For this reason it was decided to implement that feature in addition to CodeActions A.4.3. Developers can use this practical features directly in Visual Studio Code. It shows on each line how the variables have to be assigned that the proof fails.
Because the calculation of the model, how it is called in Z3, can take very long, it is not

calculated automatically if a proof fails. Nevertheless this can be overridden in the configuration.

A.4.5 Displaying Flow Graph

Something which is already implemented in Boogie, is the generation of a Flow Graph out of a boogie program. Because this could be quite useful for developers, it was decided to implement this feature as well. Mainly because it can be implemented so, that inside the IDE, the source code is on one side and the flow graph next to it. It displays the program visually including all pre- and postconditions.

A.5 Project Homepage

In order to always be fully aware about all aspects of the project, it was decided to setup a small project homepage that gathers all relevant information and displays it at a single point. This was done to support the authors with their work, but also to provide a real-time insight to the advisors of this project.
The first abstract of the page collects all links that are relevant to the project. This includes all code repositories, but also links to all project management tools used as well as to the documentation of this project.

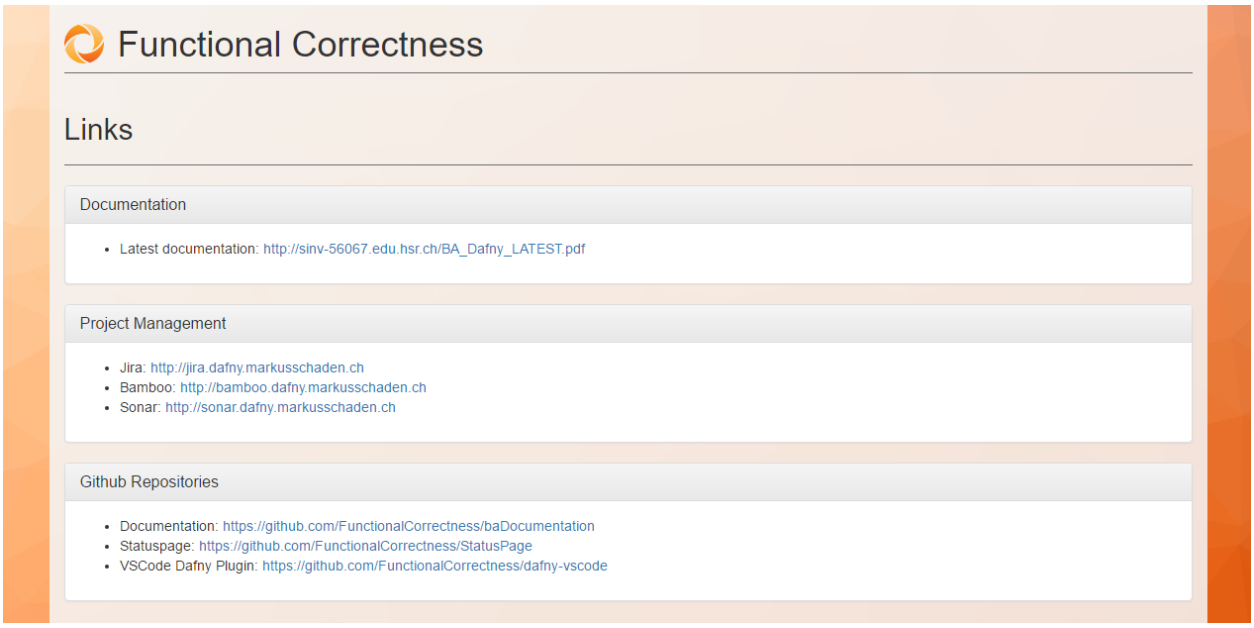

Figure A.3: All links relevant to the project

Since continuous integration was ingrained into this project from the start, the last stable version of the plugin was continuously available in the marketplace of Visual Studio Code. To keep track of how many people are already using the plugin, a counter displaying the downloads was added to the homepage.

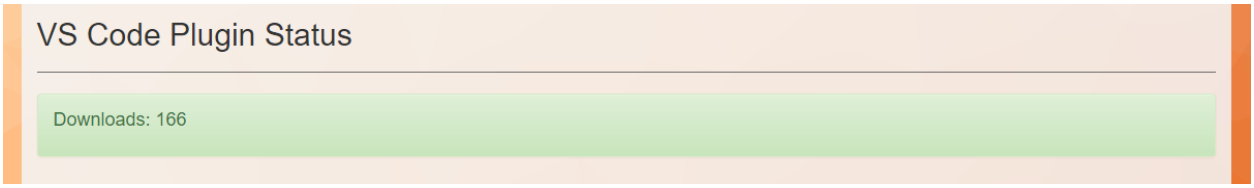

Figure A.4: Downloads of the plugin

As already mentioned, it was paramount to this project to implement continuous integration, not only with the plugin, but for all other artifacts as well (the homepage itself, or the documentation). In order to achieve this, various test- and build jobs were defined with Bamboo. To quickly see if anything is amiss, the homepage also displays the status of these tasks.

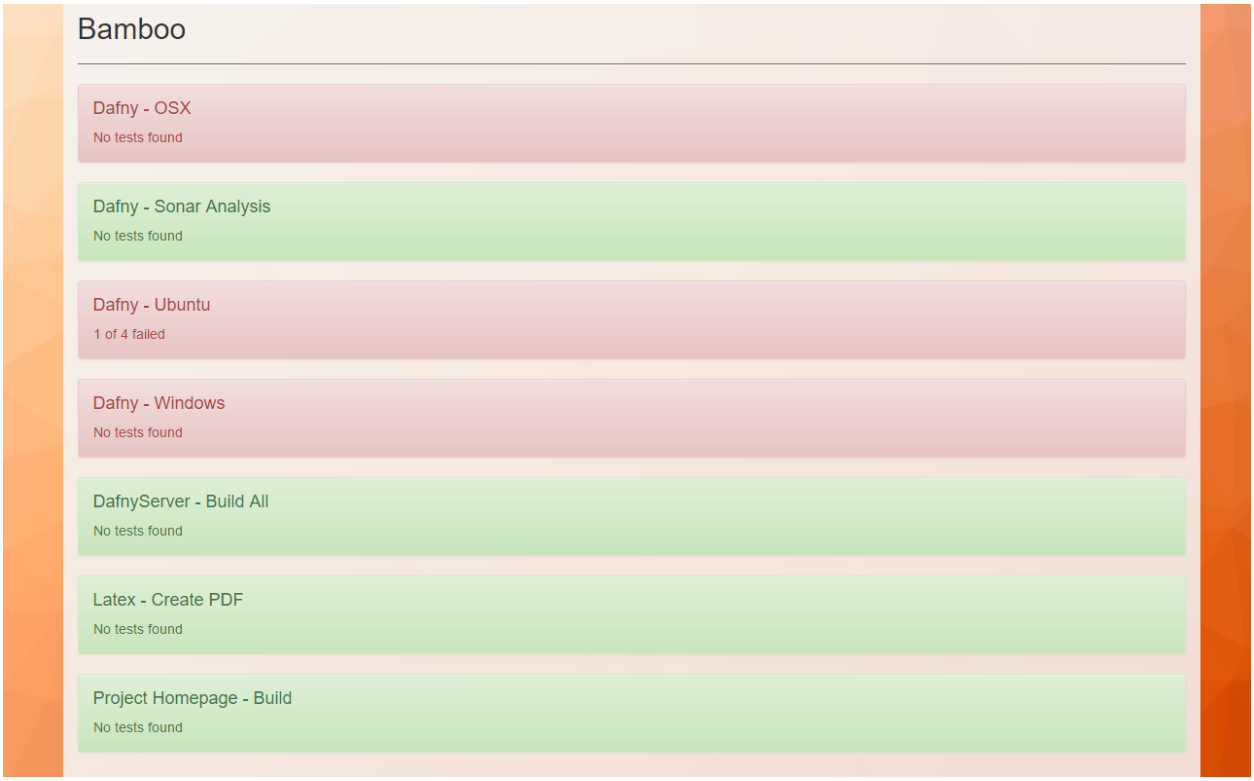

Figure A.5: Status of all automated tasks

To keep code quality at a high level, SonarQube was used to analyze every commit to the code base of the plugin. The findings of SonarQube were also integrated into the homepage.

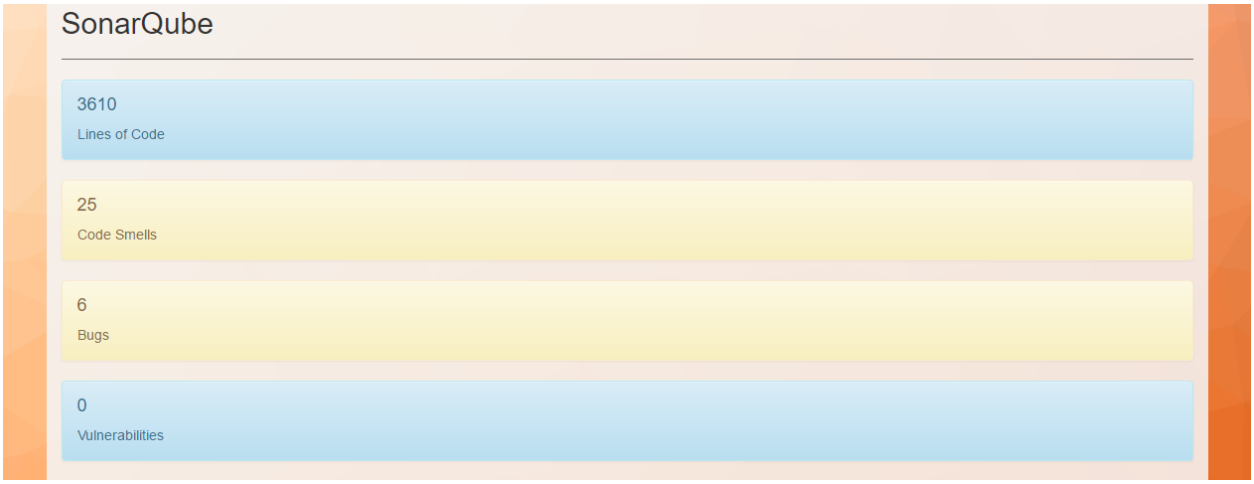

Figure A.6: Metrics analyzed by SonarQube

To support an iterative approach towards implementation, Scrum was chosen with weekly sprints. Tickets could either be new, worked upon or finished. To quickly grasp the work currently being done and how much time is planned for them, a dashboard displaying information from Jira, the project management tool used by this project, is also integrated into the homepage.

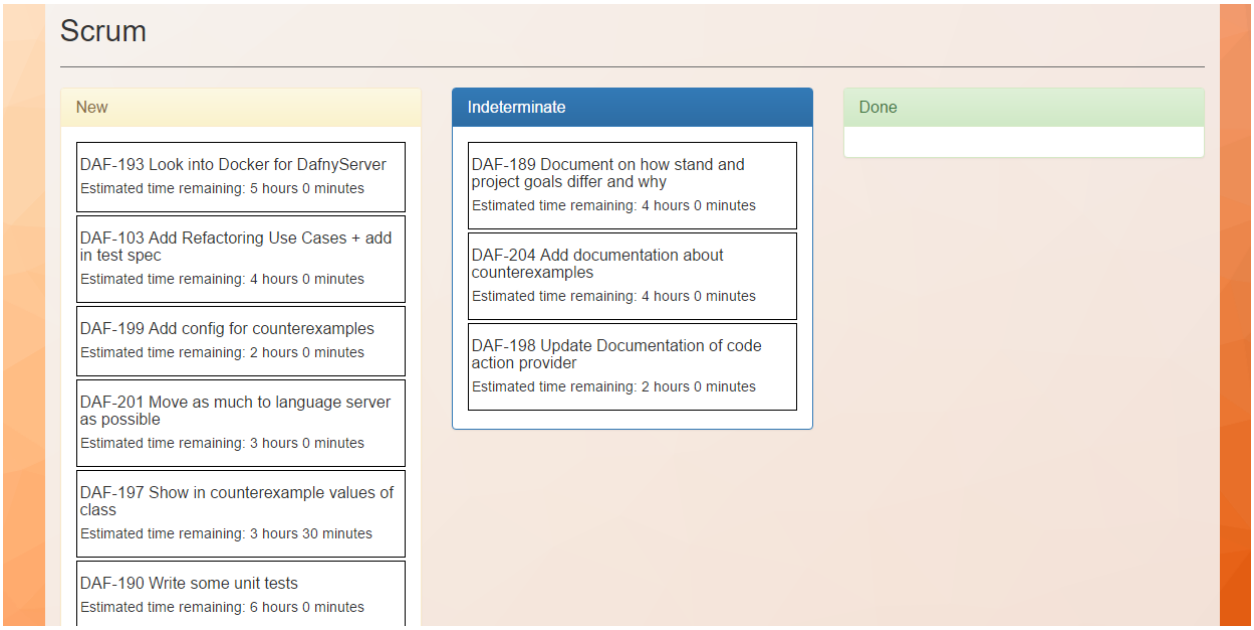

Figure A.7: Project Dashboard from Jira

Lastly, it was important to stay on track during the project regarding time management. To have a simple overview on the time invested, an additional import from Jira was done, which is implemented as graph detailing the cumulative amount of hours worked on the project.

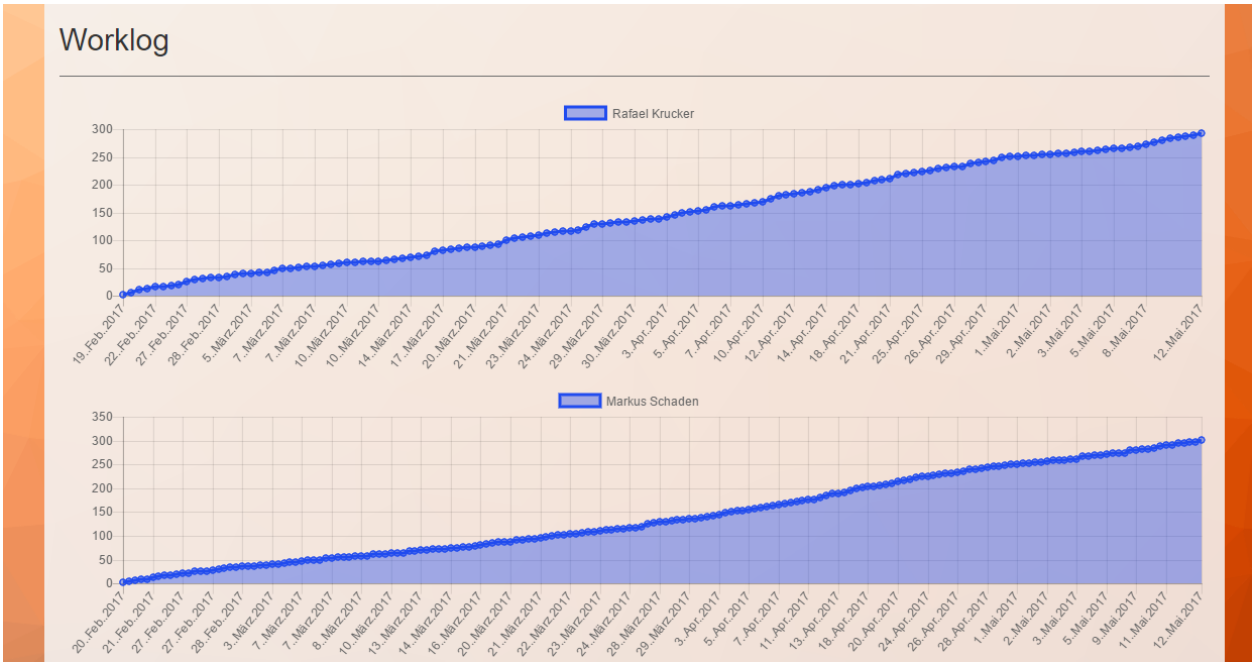

Figure A.8: Hours worked on the project

Altogether, these abstracts give a detailed overview of the project, allowing for a quick grasp of the work being done, its quality and how the project is coming along over time. It was often the first point of action to open the page when working on the project.

A.6 Time report

This chapter gives insight into how the time was invested that was spent while working on this project. First, an overview is given, which puts the different categories of work into contrast to each other. The second chart shows the progression of total work done over time. As can be seen, the total work done is well in the margin of 5% around the desired time of 720 hours which is suggested when writing a bachelor thesis and working in a team of two.

A.6.1 Time per category

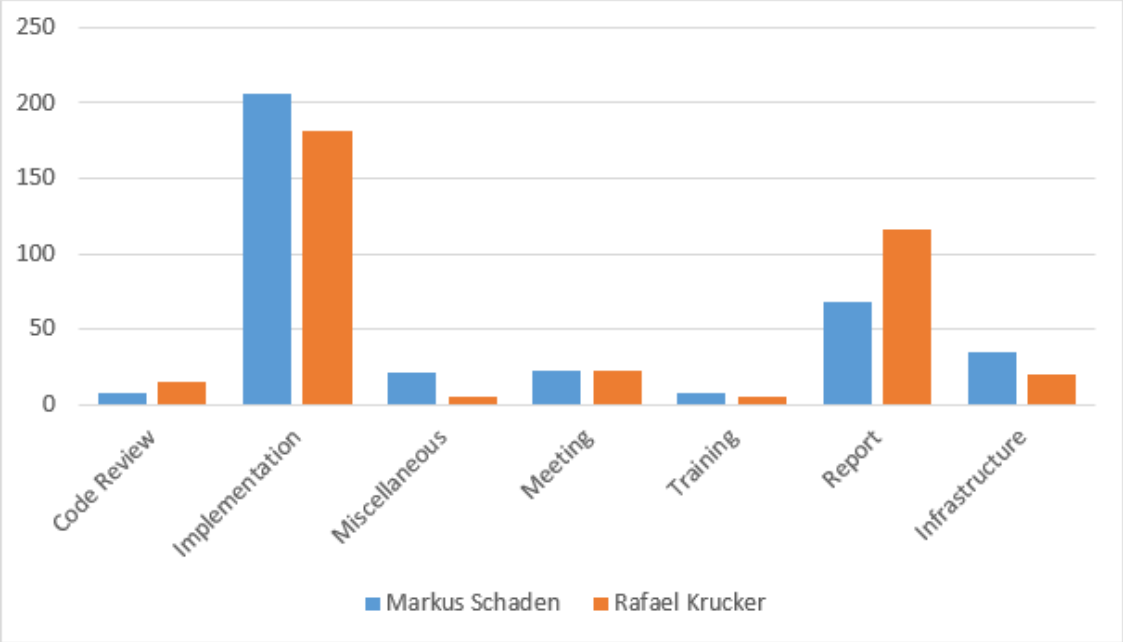

Figure A.9: Time per person per category

A.6.2 Rafael Krucker

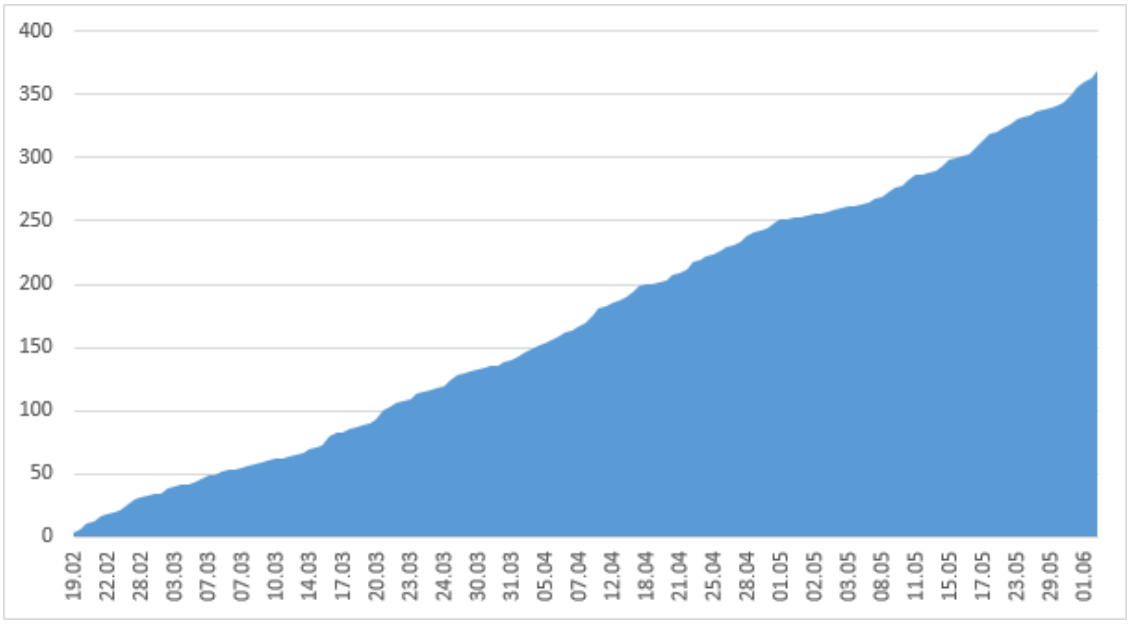

Figure A.10: Time report of Rafael Krucker

A.6.3 Markus Schaden

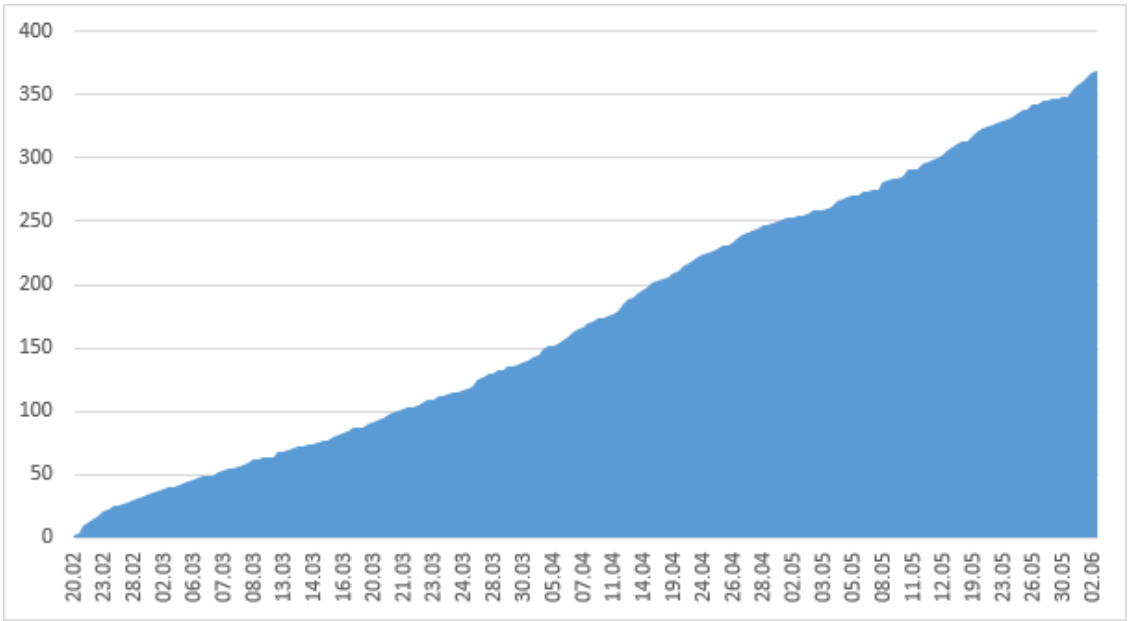

Figure A.11: Time report of Markus Schaden

A.7 Code Metrics

This project details which measures where put in place in order to ensure that the quality of the plugin followed a high standard. This was important since the goal was not to develop a proof of concept, but a production ready solution. For this project it was decided to use SonarQube, a static code metrics analyzer to gain insight into the quality of the work done. During the whole development, SonarQube was always part of the continuous integration process. Due to this, it was clear in which direction code quality developed after every commit. Very serious vulnerabilities also triggered messages being sent, so that a quick reaction was possible.
All standard configurations of SonarQube for Typescript were let be, because in this configuration SonarQube is the strictest. This meant that the coder base was subject to 117 rules regarding the code quality analysis. The only exception that had to be done was that a class called Symbol was written during the implementation. SonarQube kept getting this class confused with the built in class symbol, the usage of which is discouraged. To not be subject to false positives, it was decided to disable this rule.
Following, some key metrics delivered by SonarQube are given.

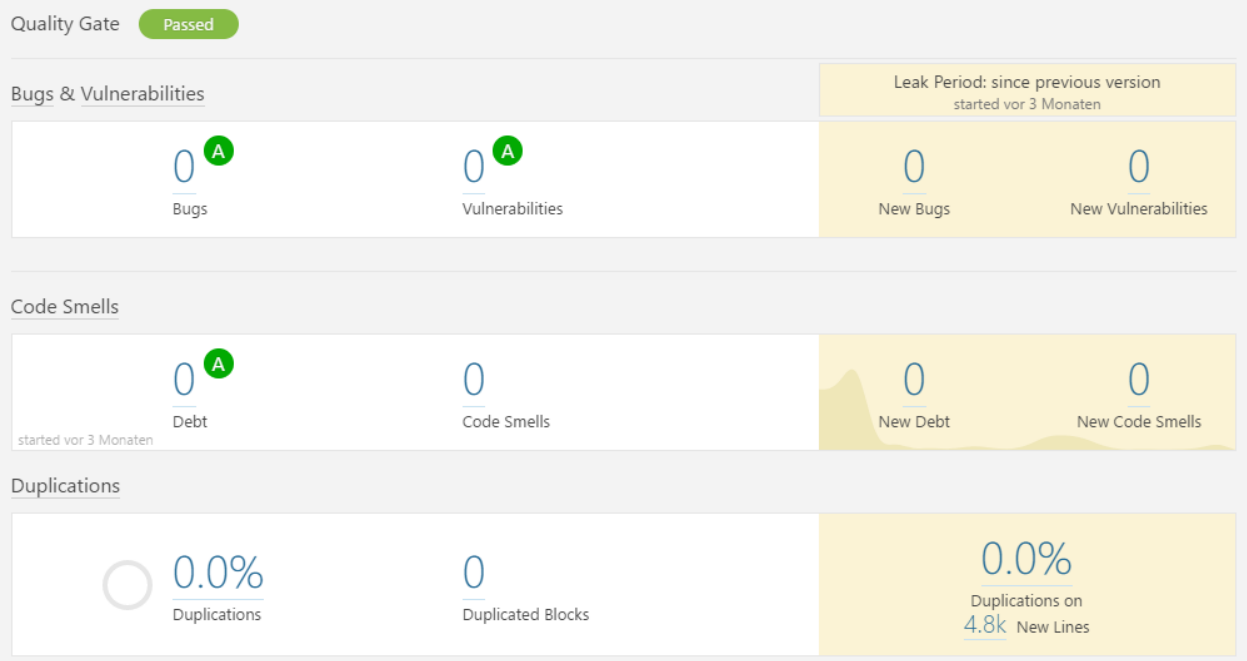

Figure A.12: Overview of the analysis

In the complete overview of the project, it can be seen that SonarQube does not detect any Bugs or Vulnerabilities in the project. Starting from about the middle of the project, all code smells that SonarQube can detect could be avoided, this due to continuous adaption of the design and the review processes. During the development a key concern was to avoid code duplication. This was done through design reviews and IDE-Tools. The analysis shows that these efforts were fruitful.
The only negative point in the way SonarQube could be integrated is that SonarQube seems

not to be able to detect code coverage generated by the application of Visual Studio Code's testing framework. After some time spent trying to make this work, it was decided to mitigate the missing information about code coverage by writing conservative test specifications which can be found in the implementation documentation.

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Coverage	Duplications
Dafny VSCode	4k	0	0	0		0.0%
client/src	800	0	0	0		0.0%
client/src/serverHelper	26	0	0	0		0.0%
server/src	308	0	0	0		0.0%
server/src/backend	802	0	0	0		0.0%
server/src/backend/features	762	0	0	0		0.0%
server/src/backend/features/codeActions	227	0	0	0		0.0%
server/src/errorHandling	43	0	0	0		0.0%
server/src/frontend	94	0	0	0		0.0%
server/src/languageDefinition	28	0	0	0		0.0%
server/src/process	76	0	0	0		0.0%
server/src/strings	198	0	0	0		0.0%
server/src/vscodeFunctions	535	0	0	0		0.0%

Figure A.13: Overview of the code base size

In the overview regarding size it can be seen that the project consists of about 4'000 lines of code. The major part thereof not surprisingly is situated in the language server part of the project. The client part consists of about 800 lines of code, but most of this is due to the tests defined here. The actual logic in the client is only about 100 lines of code, underlining the portability of this project.

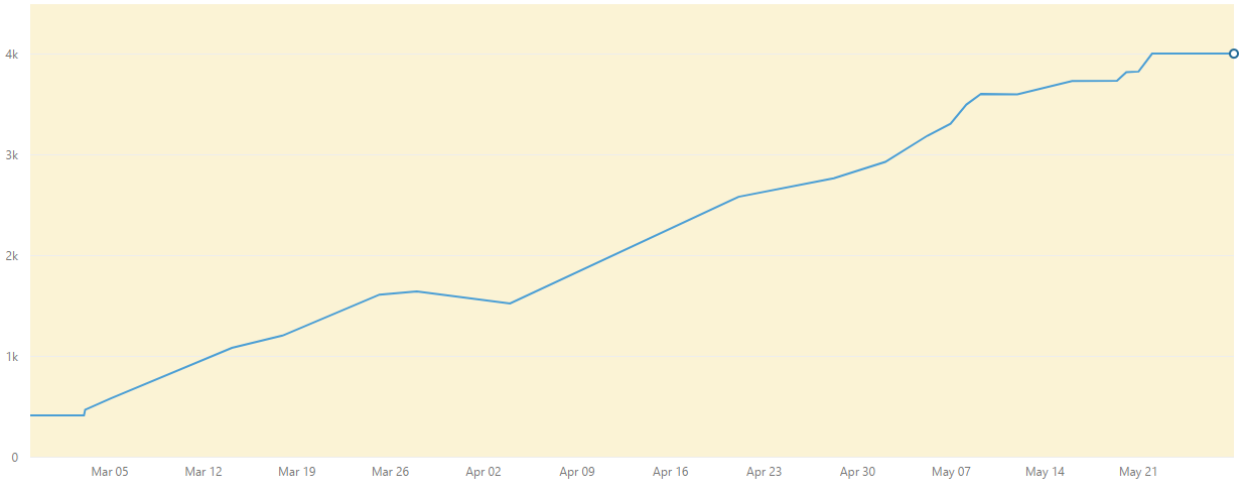

Figure A.14: Overview the amount of code written over time

As can be seen in the overview of code being added over the time, the amount of code written

steadily progressed. Through refactorings and improvements regarding the design the size was also reduced again from time to time, but new features were always quickly developed. Concluding it can be said that the analysis done by SonarQube ascertains that the project is subject to a high quality standard. The high maintainability rating and the absence of code duplication is a good indicator that this project can be continue to be developed without problems. In addition, all bugs, code smells and vulnerabilities that SonarQube can detect could be strayed away from early on in the project. It has to be said that the analysis of Typescript code is not quite as sophisticated as the one it provides for Java or C#, as for instance complexity is not analyzed in detail. However, the analysis provided lays a good basis for reasoning about the quality of the project.

B Use Cases

This project details the use cases which were defined at the beginning of this project. Over the course of this project, insights were won which had the adaption of some of the use cases as a consequence. These modifications can be read about in A.4.

B.1 Use Case Diagram

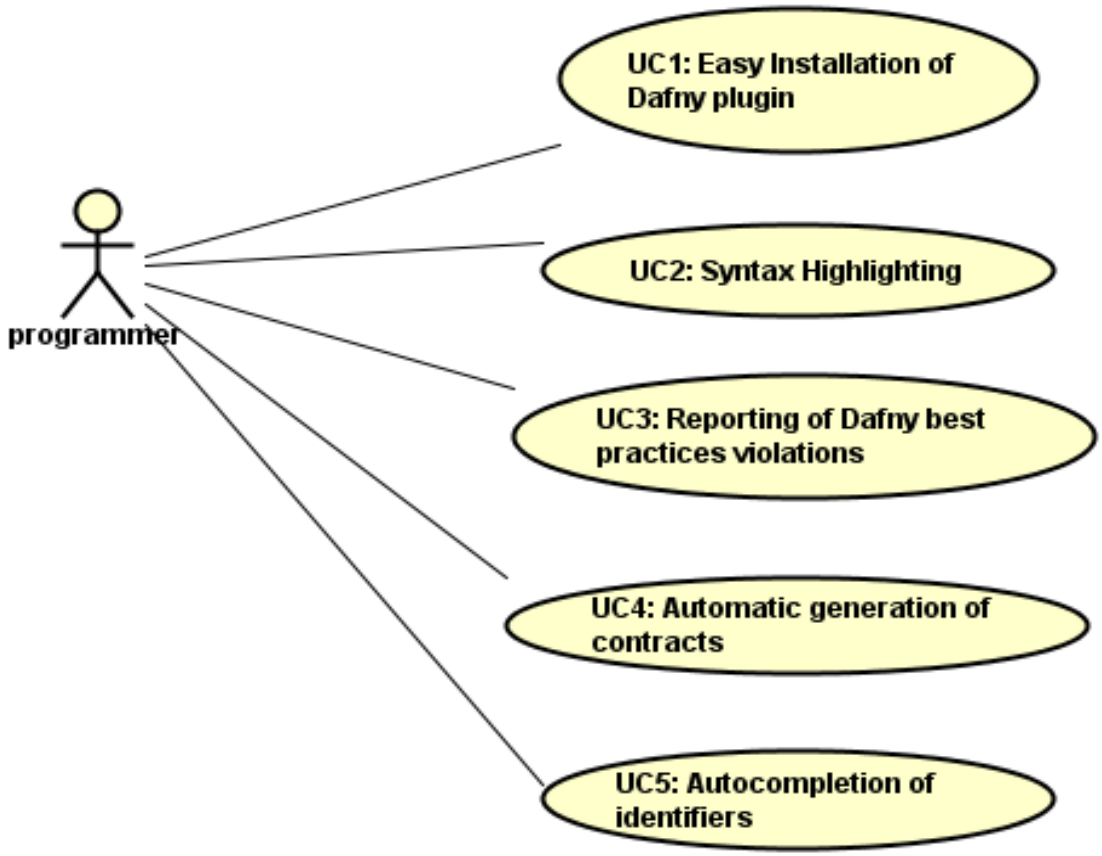

Figure B.1: Use Case Diagram

B.2 Actors and Stakeholder

- Programmers
- Microsoft

B.3 Descriptions (brief)

B.3.1 UC1: Easy installation of Dafny plugin

A programmer can simply install the Dafny plugin by running an automatic installer which sets all path variables and makes additional needed environment adjustments. This can be done on Windows 10 in a .NET environment or either on Linux or OSX in a mono environment.

B.3.2 UC2: Syntax Highlighting

The system automatically does syntax highlighting while the user writes a Dafny (dfy.) file.

B.3.3 UC3: Reporting of Dafny best practices violations

The plugin can be configured with a simple DSL config file which describes common best practices for Dafny. The plugin reports violations of these rules via the standard Visual Studio Code notification mechanisms.

B.3.4 UC4: Automatic generation of contracts

The plugin can be configured with a simple DSL-File to recognize certain situations which could benefit from the setting of pre- and postconditions. The plugin offers to add these in form of a refactoring via the common Visual Studio Code mechanisms.

B.3.5 UC5: Auto completion for identifiers

The plugin offers auto completion of Dafny code while the user types via the standard Visual Studio Code mechanisms.

B.4 Descriptions (fully dressed)

B.4.1 UC1: Easy installation of Dafny plugin

Description	A programmer can simply install the Dafny plugin by running an automatic installer which sets all path variables and makes additional needed environment adjustments.
Primary Actor	Programmer
Trigger	Programmer wants to install the Dafny plugin to Visual Studio Code.
Stakeholder and Interests	Programmer: Wants an easy, automated installation of the plugin. Microsoft: Wants a stable Dafny integration to fulfill the needs of its clients.

Preconditions	• Depending on the environment, either the .NET or the mono framework are installed. • Visual Studio Code is installed. • The programmer has admin privileges in his environment.
Postconditions	• The plugin works without problems, the programmer can start writing code.
Main Success Scenario	1. Programmer downloads the plugin via the Visual Studio Code plugin store. 2. The plugin determines which platform it is run on, and sets either the path to the .Net framework or the mono framework. 3. The plugin downloads Dafny and sets the path to the Dafny-Server binary. 4. The plugin then installs itself via the standard Visual Studio Code plugin mechanisms. 5. The plugin prompts a success message and the programmer is ready to code in Dafny.
Extensions	2a. The installer cannot find, depending on the environment, either the mono or the .Net framework in the standard locations. - It prompts the user to enter the location and does so until the framework is found.
Special Requirements	None
Frequency of Occurrence	Usually once per working environment
Open Issues	None

Table B.1: UC1

B.4.2 UC2: Syntax Highlighting

Description	The system automatically does syntax highlighting while the user writes a Dafny (.dfy) file.
Primary Actor	Programmer
Trigger	Programmer writes Dafny code in Visual Studio Code.
Stakeholder and Interests	Programmer: Wants enhanced readability for the source files he is working on. Microsoft: Wants a state of the art IDE integration of Dafny.

Preconditions	• Visual Studio Code with the Dafny plugin installed is running. • Dafny code is being written in a .dfy file.
Postconditions	• The source code is highlighted in different colors according to common standards in general and the Visual Studio Code guidelines specifically.
Main Success Scenario	1. Programmer types code into a .dfy file. 2. The plugin detects the changes and applies syntax highlighting through the standard Visual Studio Code mechanisms.
Extensions	2a. The newly written code causes a compilation error and cannot be interpreted. - The previous syntax highlighting stays in place, the errors are highlighted according to common practices with compilation errors in Visual Studio Code.
Special Requirements	None
Frequency of Occurrence	Very often, after every key up event.
Open Issues	None

Table B.2: UC2

B.4.3 UC3: Reporting of Dafny best practices violations

Description	The plugin can be configured with a simple DSL config file which describes common best practices for Dafny. The Plugin reports violations of these rules via the standard Visual Studio Code notification mechanisms.
Primary Actor	Programmer
Trigger	Programmer writes Dafny code in Visual Studio Code.
Stakeholder and Interests	Programmer: Wants to write the cleanest code possible using common Dafny idioms. Microsoft: Wants to support programmers getting the most out of Dafny.

Preconditions	• Visual Studio Code with the Dafny plugin installed is running. • Dafny code is being written in a .dfy file.
Postconditions	• Violations of common best practices for Dafny are reported through the standard Visual Studio Code mechanisms.
Main Success Scenario	1. The plugin is installed preconfigured with a collection of common best practices for Dafny, which is done through a DSL file. 2. Programmer types code into a .dfy file. 3. The plugin continuously checks for violations of the rules. 4. If a violation is detected, it is reported through the standard mechanisms of Visual Studio Code.
Extensions	1a. The predefined rules are not sufficient for the programmer. - The programmer can update the configuration file himself to include his own or his company's coding guidelines.
Special Requirements	None
Frequency of Occurrence	Very often, after new valid syntax was written.
Open Issues	None

Table B.3: UC3

B.4.4 UC4: Automatic generation of contracts

Description	The plugin can be configured with a simple DSL-File to recognize certain situations which could benefit from the setting of pre- and postconditions. The plugin offers to add these in form of a refactoring via the common Visual Studio Code mechanisms.
Primary Actor	Programmer
Trigger	Programmer writes Dafny code in Visual Studio Code.
Stakeholder and Interests	Programmer: Wants help to find appropriate contracts. Microsoft: Wants to support programmers getting the most out of Dafny.

Preconditions	• Visual Studio Code with the Dafny plugin installed is running. • Dafny code is being written in a .dfy file.
Postconditions	• Specification constructs suitable for the context were added to the code.
Main Success Scenario	1. The plugin is installed preconfigured with a collection of common situations for Dafny and their corresponding specification constraints, which is done through a DSL file. 2. Programmer types code into a .dfy file. 3. The plugin continuously checks for situations that could benefit form the setting of specification constraints. 4. If such a semantic is detected, it is reported through the standard mechanisms of Visual Studio Code, with a command offered to add the constraints. 5. The programmer invokes the refactoring and the necessary code is added.
Extensions	1a. The predefined situations and contract code additions are not sufficient for the programmer. - The programmer can update the configuration file himself to include support for his own or his company's common code semantics.
Special Requirements	None
Frequency of Occurrence	Very often upon typing code with no syntax errors.
Open Issues	None

Table B.4: UC4

B.4.5 UC5: Auto completion for identifiers

Description	The plugin offers auto completion of Dafny code while the user types via the standard Visual Studio Code mechanisms.
Primary Actor	Programmer
Trigger	Programmer writes Dafny code in Visual Studio Code.
Stakeholder and Interests	Programmer: Wants to be more productive while writing Dafny code. Microsoft: Wants a state of the art IDE integration of Dafny.

Preconditions	• Visual Studio Code with the Dafny plugin installed is running. • Dafny code is being written in a .dfy file.
Postconditions	• An identifier was auto completed.
Main Success Scenario	1. Programmer types code into a .dfy file. 2. The plugin detects the changes and searches for the beginning of known identifiers. 3. If such a beginning is found, completion if offered through the standard mechanisms of Visual Studio Code.
Extensions	None.
Special Requirements	None
Frequency of Occurrence	Very often after every key up event.
Open Issues	None

Table B.5: UC5

List of Figures

List of Tables

References

[Brü05] Stefan Brünig. *Detecting non-provable goals*. 2005.

[Dro+08] S. Drossopoulou et al. *A Unified Framework for Verification Techniques for Object Invariants*. 2008.

[Ecl17] Eclipse. *Eclipse Project*. May 2017. URL: `https://eclipse.org/`.

[Ema17] EmacsLsp. *Java Emacs Integration*. May 2017. URL: `https://github.com/emacs-lsp/lsp-java`.

[ema17] emacs-jsp. *LSP-Mode*. May 2017. URL: `https://github.com/emacs-lsp/lsp-mode/`.

[Fou17] Eclipse Foundation. *Eclipse LSP4E governance*. Apr. 2017. URL: `https://projects.eclipse.org/projects/technology.lsp4e/governance`.

[GNU17] GNU. *GNU Emacs*. May 2017. URL: `https://www.gnu.org/software/emacs/index.html`.

[Hoa09] Tony Hoare. "Null References: The Billion Dollar Mistake". In: *infoQ.com* (Aug. 2009).

[Hun99] B Hunt. *The Geometry of Some Special Arithmetic Quotients*. Springer, 1999.

[Jet17] JetBrains. *JetBrains*. May 2017. URL: `https://www.jetbrains.com/`.

[KK12] Bakhadyr Khoussainov and Nodira Khoussainov. *Lectures on Discrete Mathematics for Computer Science*. World Scientific Publishing Co Inc, 2012.

[Mey17] Bertrand Meyer. *Eiffel*. May 2017. URL: `https://www.eiffel.com/resources/faqs/eiffel-language/`.

[Mic17a] Microsoft. *Boogie*. May 2017. URL: `https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/`.

[Mic17b] Microsoft. *Monaco Editor*. May 2017. URL: `https://github.com/Microsoft/monaco-editor`.

[Mic17c] Microsoft. *The Language Server Protocol*. May 2017. URL: `https://github.com/Microsoft/language-server-protocol`.

[Mic17d] Microsoft. *Typescript in Monaco*. May 2017. URL: `https://github.com/Microsoft/monaco-typescript`.

[Mic17e] Microsoft. *Visual Studio Plugin*. May 2017. URL: `https://github.com/Microsoft/dafny/wiki/INSTALL`.

[SK13] Mohamed Nassim Seghir and Daniel Kroening. *Counterexample-guided Precondition Inference*. 2013.

[tvi17] tvi. *Sublime Integration of Dafny*. May 2017. URL: `https://github.com/tvi/sublime-dafny`.

[Z317] Z3. *Z3 Theorem Prover*. May 2017. URL: `https://github.com/Z3Prover`.