media/image1.png

Project: Dafny Language Server Redesign

Developer Documentation

Marcel Hess

Thomas Kistler

Supervisors:

Thomas Corbat

Fabian Hauser

Table of Contents

Chapter 1

Overview

This document will help you getting started to work on the Visual Studio Code Dafny plugin and the Dafny language server.

The project consists out of two main parts as you can see in Figure 1. On the one hand, there is the Visual Studio Code plugin, the so called “client”. It just contains a very basic level of logic and is mainly responsible for displaying information to the user.

On the other hand, there is the Visual Studio solution, called the “server”. It delivers the required information to the client using the language server protocol (LSP).

`media/image2.png`

Figure 1 - Architecture

Although the plugin part is called “client” and the language server “server”, please note that both instances are run on the user’s local workstation.

Chapter 2

Introductory Tutorials

This chapter presents two rather simple, but extremely helpful tutorials to get familiar with the problem domain. The first tutorial is about the creation of Visual Studio Code extensions. The second one is from OmniSharp and shows how to use their implementation of the language server protocol.

Both together are an optimal preparation for this project.

VSCode Extensions

To understand how one develops a Visual Studio Code extension, we can recommend the tutorials provided on the official site from Visual Studio Code [1]. The "Your First Extension" tutorial is very simple but you will get familiar with all important files, classes and concepts for developing an extension. Further on that site, you will find more advanced information like what kind of programmatic language features are possible with the Visual Studio Code API.

OmniSharp

OmniSharp offers the Language Server Protocol implementation for C#. Instead of starting with our project, you may first want to have a look at a more basic example of an OmniSharp implementation. "Creating a language server using .NET“ is a very well-suited tutorial for this matter [2]. It gives a nice introduction on how LSP requests are handled. Our implementation follows the same style used in the tutorial. We think it is extraordinary helpful for your understanding.

If you need further help with LSP and OmniSharp, please visit their Slack community [3].

Questions asked in the slack channel are usually answered very quickly if you keep yourself concise.

Chapter 3

Setup of the Development Environment

This chapter will provide you with a detailed introduction how to set up all necessary repositories and files, so that you can start to contribute to the project. The chapter is separated into two parts. First, we will have a look at the client side which is rather simple. To work on the client, one uses Visual Studio Code itself as an IDE. Secondly, the server side, which is quite strenuous to set up, is discussed. Some hints for CI your configuration are also given. Make sure to follow our instructions carefully. The recommended IDE for C# is Visual Studio.

For a nice start, you best create a project folder and then clone the git repository for the client as well for the server we were working on into it [4]. Please note that we renamed the folder dafny-language-server to just dafny and dafny-vscode-plugin to VSCodePlugin to keep it more simple.

Client

After you have cloned our repositories, open the folder VSCodePlugin in the client repository with Visual Studio Code. To do so, you may just open a terminal and type "code .". You can also create a batch file for a quick access to the client in Visual Studio Code [5].

Find and open the file src/extension.ts and press F5. This will launch the plugin in a virtual test environment. It can be used like it would be installed and you can set breakpoints and debug the code as well. Once your virtual environment is running you have to open a .dfy file. Otherwise the plugin will not start since the plugin listens only to Dafny files.

You should then see a notification that the server crashed. This is because we have not built it yet. We will discuss this step in chapter 3.2 - Server.

Selecting the Proper Output View

On the bottom of the screen, the console output is displayed. Often, a random window is shown. However, you are interested in the console output of "Dafny Language Server". You may have to manually choose this output window as shown in the figure below. Think of that whenever you get no output.

`media/image3.png`

Figure 2 - Set VSCode Console Output

Packages Dependencies

Although Visual Studio Code should automatically resolve all dependencies, you may need to run the "npm install" command manually. This installs all packages the client requires. Dependencies are defined in the package.json file.

Just cd into VSCodePlugin with your command prompt and run npm install.

Server

Setting up the language server is not as easy as just cloning the repository. Dafny has a bunch of dependencies which you have to provide manually. This is rather inconvenient for developers and the issue has been reported to our supervisors. Hopefully, the setup can be made easier in the future. For now, please follow the upcoming steps carefully. Please note, that the root directory in the following steps is always the created "parent folder" that contains both cloned git repositories and not the server subfolder.

Microsoft Boogie

First of all, you need to provide a lot of binaries from Microsoft Boogie [6]. Just download the repository and build the project. This should create all binaries inside the folder “boogie/Binaries”. Make sure to hit “Rebuild”, not just “Build”.

A problem that may occur is that one file, “Main.resx”, is marked as not trustworthy under Windows. Visual Studio will give you a detailed error message. To resolve this, just locate the file and allow access to it.

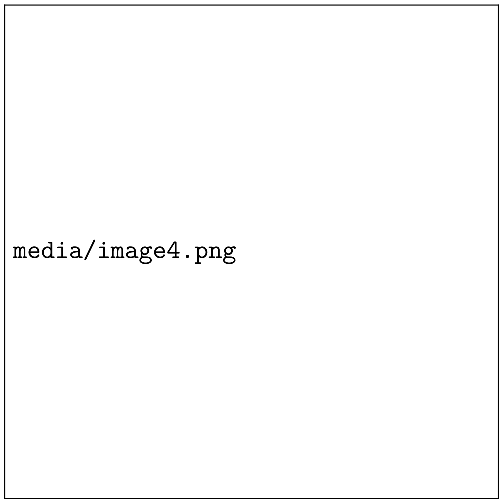

Figure 3 - Main.resx Permissions

The build should cover 21 projects and create about 40 files inside the binaries folder. Each project should have an associated .dll and .pdb file, as well as Boogie.exe itself. After building, all you need to keep is the content of the boogie/Binaries folder. Be aware that aside the .dll and .exe files, you have to keep the associated .pdb file. These refer to system libraries, such as System.Collections or such whenever a collection is used.

The other folders are technically no longer needed, but in case you want to rebuild Boogie, they may come in handy.

To provide Boogie inside your CI environment, you can as well clone the repository and build it. Our Docker file contained the following three self-explanatory lines for this purpose.

```
RUN git clone https://github.com/boogie-org/boogie.git &&\

msbuild boogie/Source/Boogie.sln

ENV PATH=$PATH:/opt/boogie/Binaries
```

If you encounter any problems during this process, Boogie has a short readme in their repository which may help you [4].

Z3

Z3.exe is a prover from Microsoft. Dafny - to be more precise: Boogie - is making use of this prover for its purposes. Dafny expects Z3.exe to be inside the dafny/Binaries/ folder. However, this exe file will not get built during the process. It is an external dependency and you have to provide the file manually. Thus, make sure that Z3.exe is located inside dafny/Binaries.

While you could clone the Z3 repository [7] and build it yourself, the process is rather inconvenient if you are not familiar with the suggested build tools like nmake. You can also simply download the current release from the release subfolder in the repository [8].

Take note that you also have to provide Z3.exe inside your CI environment. For example, we had to provide Z3 in our docker environment by downloading the file from the above repository and unzip it with the following series of commands:

```
RUN wget --no-verbose ${Z3_RELEASE} &&\

unzip z3*.zip &&\

rm *.zip &&\

mv z3* z3

ENV PATH=$PATH:/opt/z3

ARG Z3_RELEASE=https://github.com/Z3Prover/z3/releases/download/z3-4.8.4/z3-4.8.4.d6df51951f4c-x64-ubuntu-14.04.zip
```

First, the z3 release is downloaded and then unzipped. Afterwards, the zip gets deleted and everything starting with z3 (z3.exe is what we want here) is moved into the z3 directory. This directory is then provided as an environment path variable.

Visual Studio Solution Overview

Once Boogie and z3 are installed, you are ready to open the solution "Dafny.sln". It consists out of multiple projects as you can see in the figure "Dafny Solution Overview". DafnyLanguageServer is the project responsible for handling LSP requests. The other projects you find inside the Dafny folder correspond to the existing Dafny solution and remain unchanged by the scope of our project [9]. Last, there is a Test folder, which contains unit and integration tests belonging to the DafnyLanguageServer.

`media/image5.png`

Figure 4 - Dafny Solution Overview

Missing References

If you have warnings containing missing references in your solution, make sure the folder structure is correct according to chapter 3.3 - Folder Structure. It is also possible to manually refer to the corresponding Boogie dlls that you have built in the prior chapter. To clean up a missing reference, right click on "References", click "Browse" and then you can select the proper dlls, such as Provers.SMTLib.dll for example. This is shown in Figure 5 below.

Figure 5 - Setting References

References are stored in each .csproj file. You may want to inspect these with a text editor if things go wrong. Once you have all references correctly set, the solution should build.

If Visual Studio complains about nUnit references, please refer to chapter 6.3 Creating Server Tests. This is just a capitalization issue and should cause no trouble.

Testing your project setup

If you want to perform an isolated test to check if your project setup is working correctly without using the client side, you may just want to run all provided tests inside the solution. Since there are integration tests using the complete infrastructure including Z3, they are a nice indicator if your setup is working properly.

You can also just try to start the server yourself. Simply hit F5 inside Visual Studio. Maybe the launch causes already an exception, which for example happens on missing references.

Folder Structure

Make sure you provide the correct folder structure, so that you do not have to manually update any references. Inside your root folder, you should have a folder “boogie”, which contains its binaries in the subfolder “boogie/Binaries”. As well inside your root folder, you need to have a “dafny” folder containing the Dafny project with its own “Source” and “Binaries” folder as you can see in Figure 6 below.

The repository we are working on covers the necessary folder structure, but you may want to use newer distributions from Boogie. If things go wrong, you can also use the installation instructions from the “*dafny-lang*” [10] and “*boogie*” [6] repositories.

`media/image7.png`

Figure 6 - Folder Structure

Chapter 4

Debugging

Debugging is of major importance for a quick and efficient development. Since we struggled with it at first, we decided to show you in a detailed fashion how you can debug a language server project. The client is hereby not the problem. The main difficulty is how to debug the server while the client is running. As we will see, one can simply attach the debugger to the executing process. Alternative methods that we tried are also presented, including the reasons why we do not recommend them.

Client Side

As mentioned in chapter 2.1 VSCode Extensions, your client will automatically be in debug mode once you start it with F5. Client debugging should be quite straightforward.

Server Side

The client will launch the language server. Once it has started, you can attach the Visual Studio debugger to the process that started the language server. Be aware that not Visual Studio Code is starting the server, but a dedicated launch process. In a dotnet core project, one would write "dotnet DafnyLanguageServer.dll" and thus, the executing process would be dotnet. If you are in a Linux environment and you want to launch DafnyLanguageServer.exe, you probably write "mono DafnyLanguageServer.exe", and thus "mono" is the executing process. However, under a Windows environment you simply start the executable with "DafnyLanguageServer.exe", and the executing process is then "DafnyLanguageServer". How the server-process is started is defined inside the file VSCodePlugin/src/server/dafnyLanguageClient.ts. For now, we assume that we started DafnyLanguageServer.exe directly.

To be able to debug, your binaries must be up to date. You may want to always build the solution prior to debugging. Once the language server is running, select Debug -> Attach and choose the process as mentioned above and shown in Figure 7.

media/image8.png

Figure 7 - Attach Debugger to a Process

Visual Studio will afterwards switch to debug mode and you can debug as usual.

Using ReAttach

You notice yourself that many clicks are necessary to attach the debugger every single time. Thus, we recommend to use the Visual Studio Code extension ReAttch [11].

It allows you to attach the debugger using a single click, or rather a single keyboard shortcut. If your client is not ready yet, it will even wait until the selected process has started. This is a very convenient feature.

media/image9.png

Figure 8 - ReAttach in Visual Studio

Using Debug.Launch()

At first, we just used Debug.Launch(), a .NET Core system method. This will also start the debugger but you will always be prompted if you want to launch a new Visual Studio instance. Afterwards, you have to wait until the project and the debugger have loaded, which takes quite some time. Thus, this method is strongly discouraged. The best way to debug is using ReAttach.

Chapter 5

Relevant Code Information

This chapter states two important facts about the code, which are not obvious. The first subchapter is about why we have chosen to target .NET Framework instead .NET Core. The second subchapter is about the redirection of the console output stream.

Target Framework

Please note that all projects are targeting .NET Framework. While it would be favorable to use .NET Core for platform independence, not all used dependencies support the core framework. If you refer a .NET Framework project from inside a .NET Core project, system libraries will be unavailable. Thus, we decided to code in .NET Framework as well. The chosen version is .NET Framework 4.6.1.

Stream Redirection

If you take a look into the source code, you will notice that we redirect the output stream into a file.

```
string tempPath = Path.Combine(Path.GetTempPath(), "./Dafny");

Directory.CreateDirectory(tempPath);

string path = Path.Combine(tempPath, "./MsgLogger.txt");

using (StreamWriter writer =

new StreamWriter(new FileStream(path, FileMode.OpenOrCreate, FileAccess.Write)))

{

Console.SetOut(writer);
```

[. . .]

}

This has two reasons. First and important is that deeper layers from Boogie sometimes print output. We do not want side effects like these. Such console outputs can ruin the client-server connection because those logs are not valid LSP formatted outputs. Secondly, the created log can be useful for debugging in case that you develop a new feature that gets no debug information dumped by the Visual Studio Code side.

NuGet Packages Overview

In Case the NuGet Packages have to be restored, changed, updated or anything, refer to this overview. Note that if you install Omnisharp's Language Server, it will also install the dependent packages JSON.RPC and Protocol.

Project	Server	Client	JSONRPC	Protocol	Serilog
DafnyLanguageServer	X		X	X	
Test/CompileHandlerTest					
Test/CompletionHandlerTest	X				
Test/ContentManagerTests				X	
Test/CounterExampleTest					
Test/VerificationServiceTest	X				
Test/LSPIntegrationTests		X			X

OmniSharp Language Server

- DafnyLanguageServer
- Test/CompletionHandlerTest
- Test/LSPIntegrationTest
- Test/VerificationServiceTest
-

Chapter 6

Testing

This chapter gives useful hints on how to run all provided tests.

Running Client Tests

To run the client tests, you may simply cd into the VSCodePlugin folder and start with “npm run test”. However, this will not work properly since one needs to run the compile-typescript command in advance, which starts a TypeScript compiler. Usually, Visual Studio Code is compiling the code, but when running tests, you have to manually do this. Thus, you best run the following commands in order:

1. npm install
2. npm run vscode:compile-typescript
3. npm run test

The definition of the compile-typescript command can be found inside the VSCodePlugin/package.json file in case you have to make any changes.

The series of commands above are also to be used exactly the same way in your CI environment.

Running Server Tests

This chapter describes how to run server tests. Of interest is primarily the second subchapter, in which we describe how you can easily run our tests in a console-only environment, namely with your CI server.

From Inside Visual Studio

As you most likely know, server tests can directly be run from within Visual Studio. You may use Visual Studio's own test runner or the one from ReSharper, whichever is more in your favor.

`media/image10.png`

Figure 9 - Running Tests From Inside Visual Studio

From Console

If you have to start the tests from a console, you can use the nUnit Console Test Runner. It is available as a NuGet Package named "nunit.consolerunner". Using nugget, the installation is simple. The corresponding command is

nuget install nunit.consolerunner

Afterwards, just launch the nUnit runner and provide the test-project-dll as an argument. The test dlls are as well located in dafny/Binaries/. The call on our CI server looked like this:

- mono ./NUnit.ConsoleRunner.3.10.0/tools/nunit3-console.exe

VerificationServiceTest.dll

Notice that we launched the exe by using mono on the Linux based build server. Whenever you have to run a Windows executable on Linux, just type mono before the executable and you are good to go.

Creating Server Tests

We decided to use nUnit as our testing framework. This way, we are independent of mstest, which is not as simple to install on a Linux CI environment. With nUnit, we can run the tests easily from within Visual Studio, but also from within the CI server.

However, when you create a new nUnit test project as shown in Figure 10, you may notice that these target only .NET Core by default.

`media/image11.png`

Figure 10 - NUnit Test Project Wizard

Just create the project as a .NET Core project. But before you write tests, close Visual Studio and navigate to the just created project file. Open it with a text editor and change the targeted framework to “net461”. This is a bit rigorous, but works totally fine. You may also want to adjust the output path, the path where your binaries will be copied to. An example .csproj file is shown below.

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<TargetFramework>net461</TargetFramework>

<OutputPath>../../Binaries/</OutputPath>
```

```
<AppendTargetFrameworkToOutputPath>false</AppendTargetFrameworkToOutputPath>
<AppendRuntimeIdentifierToOutputPath>false</AppendRuntimeIdentifierToOutputPath>
<IsPackable>false</IsPackable>
</PropertyGroup>
<ItemGroup>
<PackageReference Include="NUnit" Version="3.11.0" />
<PackageReference Include="NUnit3TestAdapter" Version="3.11.0" />
<PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.9.0" />
</ItemGroup>
<ItemGroup>
<ProjectReference Include="..\..\Source\DafnyLanguageServer\DafnyLanguageServer.csproj"
/>
</ItemGroup>
</Project>
```

Sometimes, Visual Studio will complain that “NUnit” is not a valid package and that it cannot resolve this reference. This should not cause any trouble, but if in doubt try to rename the reference to “nUnit” with a lower n.

Test Folder Structure

Our nUnit tests are placed within the dafny/Test folder. Each project is in an own subfolder, and potential test files in another subfolder. While this makes sense so far, the previous bachelor thesis used the Test subfolder for dedicated integration tests. They scan the whole dafny/Test folder for Dafny files and expect that every .dfy file is one of their tests including a test specification file. Since our .dfy test files were not supplied with a test specification file, some errors arose.

We bypassed this problem by excluding our own folders from their test runner, but the two test concepts should actually be separated. Because we assume that the old tests, which still compare console outputs, are no longer necessary once our project is finished, we haven’t followed this issue any further.

Concept of Writing Isolated Unit Tests

This chapter covers what you should keep in mind if you write new tests or write new components that should be tested.

What to Test and What Not

Since we get requests by OmniSharp and return results from Boogie, not every implemented feature contains complex programming logic in our part of the language server backend. Instead, they just forward information and have a trivial behavior. We assume that OmniSharp and Boogie get tested well enough on their own. Therefore, if some features just forward information from OmniSharp to Boogie and back, we do not test the bypassed information.

However, if there is logic included, we did write test cases. For example, the whole class FileSymboltable, that is used by the CompletionHandler, has been covered with unit tests. In general, one can say that components which are inside the ContentManager package include more complex logic than handlers and services.

Using Interfaces for Dependency Injection

Classes with deeper dependencies to Boogie or OmniSharp support dependency injection. For example, this is the case in the class FileSymbolTable:

```
private IDafnyTranslationUnit _translationUnit;
public FileSymboltable(IDafnyTranslationUnit translationUnit)
{
_translationUnit = translationUnit;
_symbolTable = GetSymbolList();
}
```

As you can see in the code snippet above, we defined interfaces like IDafny-TranslationUnit. This interface is implemented by DafnyTranslationUnit for the original use, but for example also by DafnyTranslationUnitFakeForCompletions. This fake has been created to write isolated unit tests for components that are used by features like autocompletion.

If you cannot write simple unit tests for your code because of heavy dependencies, you can probably refactor your code and use dependency injection. Once you use proper interfaces, it should be easy to write a mock or a fake that allows writing isolated unit tests with ease.

Chapter 7

Continuous Integration and Continuous Delivery (CI/CD)

Our CI/CD is already linked to GitLab. In case one develops further changes, it works out of the box. In case you wish to reconfigure for your own GitLab repository, you will find helpful hints in this chapter. For a detailed description of the used GitLab stages, please follow the main document.

Updating Versions

The Docker image includes several third party dependencies like Z3 for Dafny, Node for the client and a Sonar scanner. You can find them in the Dockerfile.build file. They are listed at the very beginning of the file as ARGs as shown in the following code snipped. The versions are clearly recognizable and easy to change.

ARG NODE_VERSION=10.16.3

ARG Z3_RELEASE=https://github.com/Z3Prover/z3/releases/download/z3-4.8.4/z3-4.8.4.d6df51951f4c-x64-ubuntu-14.04.zip

ARG GO_RELEASE=go1.10.3.linux-amd64.tar.gz

ARG SonarScanner_RELEASE=3.0.3.778

Please note that when the Dockerfile.build file changes, the next CI pipeline will take longer since the prebuild stage for building the docker container will be triggered again.

Prebuild Stage

As mentioned in the previous chapter, the prebuild stage runs every time the Dockerfile.build file is changed. The same applies whenever the yaml configuration file is edited. This is defined by the changes tag in the following snippet from .gitlab-ci.yml:

```
build_image:
stage: prebuild
image: docker:latest
only:
refs:
- master
changes:
- Dockerfile.build
- .gitlab-ci.yml
```

As you can notice, there is also a refs tag in the only section. It means that the docker image gets only built when the changes happen on the master branch. If you would like to change this behavior, you have to update the .gitlab-ci.yml file.

Adjusting the Sonar Token

If you would like to link the repository to another SonarCloud project for code metrics, you have to change the SONAR_TOKEN in GitLab. To do so go to Settings > CI/CD and expand “Variables”. Now you can add or change the environment variable as shown in Figure 11 below. To generate a new token, please follow the SonarCloud user guide [12].

media/image12.png

Figure 11 - Add SONAR_TOKEN as Environment Variable

Chapter 8

References

[1] "Extension API." [Online]. Available: https://code.visualstudio.com/api/index. [Accessed: 14-Oct-2019].

[2] "Martin Björkström - Creating a language server using .NET." [Online]. Available: http://martinbjorkstrom.com/posts/2018-11-29-creating-a-language-server. [Accessed: 14-Oct-2019].

[3] "OmniSharp community on Slack." [Online]. Available: https://omnisharp.herokuapp.com/. [Accessed: 14-Oct-2019].

[4] "Dafny SA / Dafny_Server_Redesign," *GitLab*. [Online]. Available: https://gitlab.dev.ifs.hsr.ch/dafny-sa/dafny-server-redesign. [Accessed: 11-Dec-2019].

[5] "launch_VScode.bat," *GitLab*. [Online]. Available: https://gitlab.dev.ifs.hsr.ch/dafny-sa/dafny-server-redesign/blob/master/VSCodePlugin/__launch__VScode.bat. [Accessed: 11-Dec-2019].

[6] "boogie-org/boogie," 20-Oct-2019. [Online]. Available: https://github.com/boogie-org/boogie. [Accessed: 22-Oct-2019].

[7] "Z3Prover/z3," 11-Dec-2019. [Online]. Available: https://github.com/Z3Prover/z3. [Accessed: 11-Dec-2019].

[8] "Z3Prover Releases," *GitHub*. [Online]. Available: https://github.com/Z3Prover/z3. [Accessed: 11-Dec-2019].

[9] "dafny-lang/dafny," 11-Oct-2019. [Online]. Available: https://github.com/dafny-lang/dafny. [Accessed: 14-Oct-2019].

[10] "dafny-lang installation," *GitHub*. [Online]. Available: https://github.com/dafny-lang/dafny/wiki/INSTALL. [Accessed: 28-Oct-2019].

[11] “ReAttach - Visual Studio Marketplace.” [Online]. Available: https://marketplace.visualstudio.com/items?itemName=ErlandR.ReAttach. [Accessed: 22-Oct-2019].

[12] “User Token | SonarCloud Docs.” [Online]. Available: https://sonarcloud.io/documentation/user-guide/user-token/. [Accessed: 11-Dec-2019].