Extending Dafny to concurrency

Owicki-Gries style program verification for the Dafny program verifier

Denissen, P.E.J.G.

Award date:
2017

Extending Dafny to Concurrency

Owicki-Gries style program verification for the Dafny program verifier

Philippe Denissen

Supervisors:
Kees Huizing
Ruurd Kuiper

Master Thesis

Software Engineering and Technology, Computer Science
Eindhoven University of Technology
The Netherlands
October, 2017

Contents

Part I

Combining Dafny and Concurrency

Chapter 1

Introduction

1.1 Motivation

Writing parallel programs is hard and error prone. Producing correct sequential programs is already difficult, but with program components operating in parallel the program complexity rapidly increases, along with the number of possible execution paths. Testing parallel programs can help to identify problems, but it may not detect all issues, in particular if the number of possible execution paths becomes infinite. This calls for formal program verification, i.e. to show that for any given input the program adheres to its intended behaviour. There are two different approaches: for finite state systems there is the model-checking approach and for infinite state systems there is the theorem proving approach. We choose the latter.

The Owicki-Gries theorem proving approach shows most directly the essentials of verification of parallel pograms using an assertional approach. It is used in education to explain parallel program verification. Our focus lies on the educational aspect, specifically to support the Owicki-Gries approach with tooling. Dafny provides good tool support for sequential verification and is already being used to teach the sequential part of program verification. Preliminary work has been done prior to this master's thesis to successfully pilot a separate transpiler tool to show that even though the Owicki-Gries method reasons about parallel programs, the resulting proof obligations are sequential and can thus be proven using Dafny.

In this master thesis we propose an extension that embeds Owicki-Gries support directly into Dafny with the goal to provide students with automated support to both support their understanding of parallel verification as well as improve the efficiency of and prevent human mistakes in producing correctness proofs.

1.2 Relevance and additional motivation of choice of Owicki-Gries

Although the approach already dates from 1976 [27] it is still widely used in universities and institutes that apply the Owicki-Gries approach (practical or in education).

In Appendix A we give a list of some such universities and institutes.

There are various other theorem approaches to verify parallel programs. For example the more modular rely-garantuee approaches [9, 25, 26], and the powerful separation logic approaches [3, 4, 11]. However these approaches are more sophisticated and therefore less suitable in an educational setting.

Owicki-Gries makes it possible to split the verification of local correctness and global correctness, but does not achieve full modularity. In our opinion the simplicity and intuitiveness of the approach outweighs this drawback.

Chapter 2

Owicki-Gries

We briefly discuss the theory of Susan Owicki and David Gries for parallel program verification [27], which assumes an interleaving concurrency model. A program contains program components that execute in parallel. Statements inside a program component are executed in order, but may be interleaved by the execution of statements from other program components.

If the program starts in a state that satisfies the precondition then, after all its program components have finished execution, the postcondition will hold regardless of the possible interleavings of the statements belonging to each of the different program components.

To prove correctness of such a program, we need to show the following:

(1) Initialization

(2) Finalization

(3) Local Correctness

(4) Global Correctness

(1) Initialization

The program's precondition implies each of the program components' preconditions.

(2) Finalization

All of the program components' postconditions together imply the program's postcondition.

(3) Local Correctness

For every program component, each of its Hoare-triples $\{P\}s\{Q\}$ are valid, where s is a statement, P is the precondition of s and Q is the postcondtion of s.

(4) Global Correctness

For every program component, each of its assertions a is *interference-free* with all statements s of the other program components, i.e. for every statement s in all other program components the Hoare-triple $\{a \wedge P\}s\{a\}$ is valid, where P is the precondition of s.

Example

Consider the following parallel program specification:

```
          { x == 0 }
[ x := x + 1 || x := x + 2 ]
          { x == 3 }
```

With the following standard proof outline:

```
                    { x == 0 }
{ x == 0 \/ x == 2 }    { x == 0 \/ x == 1 }
     x := x + 1              x := x + 2
{ x == 1 \/ x == 3 }    { x == 2 \/ x == 3 }
                    { x == 3 }
```

Initialization : we need to show that the following two implications hold:

$$x == 0 \implies x == 0 \vee x == 2 \tag{2.1}$$

$$x == 0 \implies x == 0 \vee x == 1 \tag{2.2}$$

Finalization : we need to show that the following implication holds:

$$(x == 1 \vee x == 3) \wedge (x == 2 \vee x == 3) \implies x == 3 \tag{2.3}$$

Clearly, the implications in Equations 2.1, 2.2 and 2.3 are true.

Local Correctness : we need to show the that following two Hoare triples are valid:

```
{ x == 0 \/ x == 2 }
     x := x + 1
{ x == 1 \/ x == 3 }
```

Listing 2.1: Local Correctness: Left Program Component

```
{ x == 0 \/ x == 1 }
      x := x + 2
{ x == 2 \/ x == 3 }
```

Listing 2.2: Local Correctness: Right Program Component

Indeed, the Hoare triples in Listing 2.1 and 2.2 are valid.

Global Correctness : we need to show that the following four Hoare triples are valid:

```
{ (x == 0 \/ x == 2) /\ (x == 0 \/ x == 1) }
      x := x + 2
{ x == 0 \/ x == 2 }
```

Listing 2.3: Global Correctness: Left Program Component - Assertion 1

```
{ (x == 1 \/ x == 3) /\ (x == 0 \/ x == 1) }
      x := x + 2
{ x == 1 \/ x == 3 }
```

Listing 2.4: Global Correctness: Left Program Component - Assertion 2

```
{ (x == 0 \/ x == 1) /\ (x == 0 \/ x == 2) }
      x := x + 1
{ x == 0 \/ x == 1 }
```

Listing 2.5: Global Correctness: Right Program Component - Assertion 1

```
{ (x == 2 \/ x == 3) /\ (x == 0 \/ x == 2) }
      x := x + 1
{ x == 2 \/ x == 3 }
```

Listing 2.6: Global Correctness: Right Program Component - Assertion 2

The Hoare triples in Listing 2.3, 2.4, 2.5 and 2.6 are all valid, and thus all assertions are interference-free.

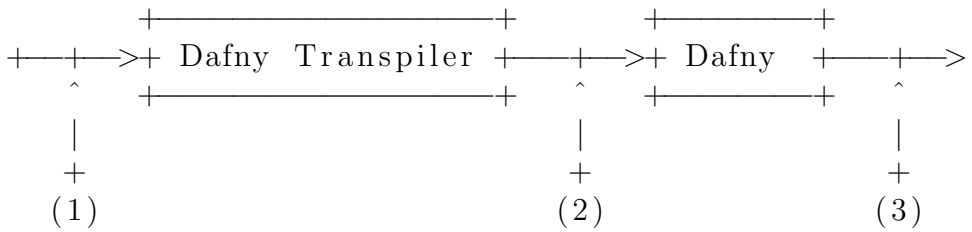
Chapter 3

Preliminary Work

3.1 Dafny Transpiler

3.1.1 Motivation

The idea to use Dafny for verifying concurrent programs in the Owicki-Gries style came from the following observation made by Kees Huizing: While the Owicki-Gries method allows us to reason about parallel programs, the proof obligations required by the method are all sequential and consist only of Hoare-triples and logical implications. Dafny is already able to prove such obligations. Concretely, once a proof outline has been established, the proof obligations follow directly from the program's precondition, postcondition and the parallel components' preconditions, postconditions and assertions. This is a syntactic transformation that can be performed in an automatic fashion. We call the tool that performs this tranformation the Dafny Transpiler. Together with Dafny, this enables the verification of (simple) parallel programs using the Owicki-Gries method, see also Listing 3.1.



Listing 3.1: Dafny Transpiler Pipeline

(1) The transpiler accepts a specially annotated parallel (syntactically correct Dafny) program, e.g. Listing 3.2.

(2) The transpiler generates a Dafny program corresponding to the specified Owicki-Gries requirements, e.g. Listing 3.3.

(3) Dafny verifies the generated program as correct if and only if the original specially annotated parallel program from (1) is correct.

3.1.2 Explanation

The transpiler takes syntactically correct annotated Dafny code as input and produces verifiable Dafny code as output. A properly annotated and commented correct proof outline yields a Dafny program that is correct if and only if the intended Owicki-Gries is correct. It assumes a parallel program written as follows:

- a `class` that encompasses the parallel program
- zero or more `instance variables` used by the parallel program components
- one `method`, typically called `Main`, to describe the parallel program, with the following properties:
- one assertion for the precondition, ending with a `// #POG` annotation
- a `// #parallel` annotation to start the list of parallel program components
- two or more `method` calls corresponding to the parallel program components
- an `// #endparallel` annotation to end the list of parallel program components
- one assertion for the postcondition, ending with a `// #QOG` annotation
- two or more `methods` that describe the parallel program components listed in the `Main` method. Each statement (or multiple statements, if they are atomic), is surrounded with `assertions`:
- the first `assertion`, the precondition of the statement, is annotated by appending `// #POG`
- the second `assertion`, the postcondition of the statement, is annotated by appending `// #QOG`
- optionally, to avoid duplicating the `// #QOG` and `// #POG` assertions, the `// #ROG` annotation can be used to annotate `assertions` between sequential non-atomic statements

To help the parser with correctly identifying methods, each method is closed by annotating the line with the closing curly bracket with `// #endmethod MethodName`.

An example of such an annotated Dafny program is shown in Listing 3.2.

```
// Use Owicki-Gries.

class Program
{
  var x: int;

  method Main()
  {
    assert x == 0; // #POG
    // #parallel
    ProgramComponentA();
    ProgramComponentB();
```

```
    // #endparallel
    assert x == 3; // #QOG
  } // #endmethod Main

  method ProgramComponentA()
  {
    assert x == 0 || x == 2; // #POG
    x := x + 1;
    assert x == 1 || x == 3; // #QOG
  } // #endmethod ProgramComponentA

  method ProgramComponentB()
  {
    assert x == 0 || x == 1; // #POG
    x := x + 2;
    assert x == 2 || x == 3; // #QOG
  } // #endmethod ProgramComponentB
}
```

Listing 3.2: Transpiler Input: a specially annotated Dafny program

The transpiler is intended as a proof of concept and as such, it has (severe) limitations. A possibly incomplete list is given below:

- The transpiler uses a primitive parser made out of regular expressions. It uses heuristics to find the information it needs to generate Owicki-Gries proof obligations.
- The transpiler generates code that Dafny can verify. It does not generate code that is intended to run as a program, verification is its only goal.
- The `Main()` method needs to be defined before the program components it contains.
- the `Main()` method should only contain one parallel block and cannot contain statements before or after that parallel block.
- Program components cannot have parameters in their method declaration.
- Program components cannot use `requires` and `ensures` in their method annotations. Instead, assertions are used.
- Program components can use a `decreases *;` in their method annotation when their body is possibly non-terminating. This method annotation will be added to the its local correctness methods, as well as each global correctness method that contains a statement of the non-terminating method.
- Everything between two assertions should be one (or more) valid Dafny statements. For example, the following will result in invalid Dafny code, because `if (myGuard) {` is not a valid statement:

```
// ...
assert 'foo'; // #POG
if (myGuard) {
```

```
    assert 'bar'; // #QOG
  }
  // ...
```

- Everything between two assertions is one atomic statement. It is recommended to use Dafny's multi-assignment statement.

Eespecially important is the last remark in the list. The Owicki-Gries method requires that every atomic statement is surrounded by assertions. Consequently, the transpiler generates Dafny code with Owicki-Gries proof obligations under the assumption that everything between two assertions is atomic. For example, a loop between two assertions means that the complete loop will be one atomic statement, from start to finish, without any possible interleaving between iterations of the loop.

The Dafny program generated by the Dafny Transpiler with the annotated Dafny program from Listing 3.2 as input is shown in Listing 3.3.

```
// Automatically generated by the Dafny Owicki-Gries Transpiler.

class Program
{
  var x: int;

  // Initialization

  method InitializationCorrectness()
    requires x == 0;
    ensures x == 0 || x == 2;
    ensures x == 0 || x == 1;
  {
  }

  // Local Correctness

  method LocalCorrectnessProgramComponentA1()
    modifies this;
    requires x == 0 || x == 2;
    ensures x == 1 || x == 3;
  {
    x := x + 1;
  }

  method LocalCorrectnessProgramComponentB1()
    modifies this;
    requires x == 0 || x == 1;
    ensures x == 2 || x == 3;
  {
    x := x + 2;
  }
```

```
// Finalization

method FinalizationCorrectness()
  requires x == 1 || x == 3;
  requires x == 2 || x == 3;
  ensures x == 3;
{
}

// Global Correctness

// R: ProgramComponentA --- S: ProgramComponentB
method GlobalCorrectness1()
  modifies this;
  requires x == 0 || x == 1;
  requires x == 0 || x == 2;
  ensures x == 0 || x == 1;
{
  x := x + 1;
}

// R: ProgramComponentA --- S: ProgramComponentB
method GlobalCorrectness2()
  modifies this;
  requires x == 2 || x == 3;
  requires x == 0 || x == 2;
  ensures x == 2 || x == 3;
{
  x := x + 1;
}

// R: ProgramComponentB --- S: ProgramComponentA
method GlobalCorrectness3()
  modifies this;
  requires x == 0 || x == 2;
  requires x == 0 || x == 1;
  ensures x == 0 || x == 2;
{
  x := x + 2;
}

// R: ProgramComponentB --- S: ProgramComponentA
method GlobalCorrectness4()
  modifies this;
  requires x == 1 || x == 3;
  requires x == 0 || x == 1;
  ensures x == 1 || x == 3;
{
```

```
    x := x + 2;
  }
}
```

Listing 3.3: Transpiler Output: a Dafny program

Listing 3.3 contains all the Owicki-Gries proof obligations we showed by hand in Section 2.

3.1.3 Field Test

The Dafny Transpiler has been used in the course Program Verification Techniques. In general, students appreciated being able to verify certain proofs automatically rather than do it by hand. We spoke to students who followed the course after we demoed the Dafny Transpiler at a lecture, interactively with the audience. Some indicated the tool helped them get a better understanding of the Owicki-Gries method because it allowed them to rapidly change their proofs and try out different strategies.

Chapter 4

Literature

4.1 Existing Verification Tools

Support for Owicki-Gries has been added to several theorem provers.

4.1.1 Isabelle/HOL

Isabelle [1] is a proof assistant and theorem prover. Mathematical expressions and formulas can be expressed in a formal language, which can be proven by the tool using logical calculus. Isabelle/HOL provides higher-order logic support to the proof system. [26] has formalized the Owicki-Gries method in Isabelle/HOL, by defining the abstract syntax, operational semantics and showing soundness of the approach.

4.1.2 PVS

PVS [2] is a verification system with a specification language and a theorem prover. It uses typed higher-order logic to specify programs, which can be proven by the theorem prover. [23] show a method to support assertion-based proving techniques (like Owicki-Gries) to PVS, such that automated strategies can show correctness of the proof outlines.

4.2 Implementation Targets

There are many theorem provers and verification systems that could be a possible candidate for our Owicki-Gries implementation. Our goal is to provide an environment which can be used in an education setting to prove parallel programs in the Owicki-Gries style of program verification. We will briefly discuss several candidates and ultimately decide to use Dafny as our implementation target.

[1] https://isabelle.in.tum.de/
[2] http://pvs.csl.sri.com/

4.2.1 KeY

KeY [3] is a program verification tool that offers software verification of programs written in Java based on symbolic execution and invariant reasoning [4] by annotating programs with their formal specification. A methodology to verify concurrent Java programs with KeY is developed by [22]. Java as a language has too many intricacies to be a viable language for an Owicki-Gries style verification aimed at teaching concurrent program verification.

4.2.2 VerCors

VerCors [5] [4] is a verification tool that acts as a compiler to translate annotated programs to a simpler language. For example, it can take a Java program as input and use Chalice as its back-end for concurrent programs, or Boogie for sequential programs. As mentioned at the end of Section 4.2.1, we dismiss Java as a viable language for our Owicki-Gries implementation with teaching concurrent program verification as its goal.

4.2.3 Boogie

Boogie [2] [14] [13] is an Intermediate Verification Language (IVL) that can express many different language features and verification techniques. Other languages can reuse Boogie's verification condition generation by encoding the necessary proof obligations ad Boogie programs. This is what Dafny does internally and our implementation of the Dafny extension provides the encoding to support Owicki-Gries style parallel programs, see also Section 7. Boogie is very powerful as an IVL and could be considered as a low level verfication language, which makes it unsuitable as the main target for our implementation.

An extension to Boogie proposes the Concurrent Intermediate Verification Language (CIVL) [10] that supports reasoning about concurrent programs. CIVL is very powerful, but has many features that are not needed for the Owicki-Gries style program verification, which makes it too complex as an implementation target.

4.2.4 Viper

Viper [25] is a verification infrastructure for permission-based reasoning. It aims to solve the shortcomings of an IVL like Boogie, namely the unsuitableness for supporting techniques based on permission-based logics such as separation logic. The Viper verification structure provides native support for the the permission-based reasoning model. The Owicki-Gries style of verification is not permission based, so the Viper infrastructure offers us no immediate benefits.

[3] https://www.key-project.org/
[4] http://www.envisage-project.eu/key-deductive-verification-of-software/
[5] http://fmt.cs.utwente.nl/research/projects/VerCors/

4.2.5 Chalice

Chalice [18] is a language and program verifier aimed at concurrent programs. It takes an annotated program and checks if the annotations are never violated. Concurrency in Chalice uses permission based reasoning and sharing objects happens via monitors to ensure suitable mutual exclusion. It is built upon Boogie, which is used to prove the generated verification conditions. Since Chalice is based around concurrency, there is no added value in implementing an Owicki-Gries style verification since it already has the ability to verify parallel programs.

4.2.6 Dafny

Dafny [16] is a verification language and a program verifier. The Dafny language is imperative and object-based, which makes it a very suitable target for education, in particular for students exposed to programming languages like Java in their curriculum. Dafny has been used successfully to teach program verification in the Master course Program Verification Techniques at the Computer Science and Engineering department of the Eindhoven University of Technology. Its ease of use and familiarity to the students makes Dafny an excellent target for our Owicki-Gries implementation, allowing students to use one tool to show corrrectness of sequential programs and simple concurrent programs.

Chapter 5

Additional Tooling for Dafny: Atom

5.1 Importance

There are several aspects of a program verifier that contribute to the user experience of working with such a tool. At the heart lies the logic system, on top of that the verifier's automation mechanisms and proof strategies, and finally the interface between the tool and the users. Dafny offers the complete package and for the latter users are provided with the Dafny Integrated Development Environment (Dafny IDE)[19].

The Dafny IDE is excellent and provides many advanced features to truly understand what is going on internally within a proof, such as integrated debugging to see what is happening on a Boogie level by providing a tight coupling with the Boogie Verification Debugger [13]. During the development of advanced Dafny programs with more involved and complicated proofs, such detailed feedback from the underlying proof system can be paramount to truly understand what is happening at the logic level and what is needed to complete the proof. The main objectives of this master project is to provide the students with automated support for understanding the parallelism features of the Owicki-Gries approach. For such an audience that is learning how to verify (parallel) programs, such detail is not needed. Examples and assignments will be constructed in such a manner that the basic concepts of constructing a proof will be taught, without needing to be privy to the intricate details of the underlying proof system.

5.2 Motivation

The existing Dafny tooling provides either too much or too little, so we provide a middle way. The Dafny IDE is an extension of Microsoft Visual Studio, an advanced programming environment with support for various high-level programming languages, like C++. Unfortunately, Microsoft Visual Studio aims for the Windows operating system and students may be using something else. It also has a large

footprint and not every student may feel inclined to install such a large application for only one class. There is an alternative to running Dafny in Visual Studio, it can also be run from a web browser [1]. This allows to get started with Dafny without even having to install Dafny itself, but lacks the comfort of editing programs locally in an editor with superior text editing capabilities.

5.3 Atom

We chose Atom [2] as our target environment. Atom profiles itself as a 'hackable text editor for the 21st century' and is developed by GitHub [3], a company with one of the largest communities to share, develop and engage in open source software. Atom is completely open source and is licensed under the MIT License, allowing anyone to modify and contribute to its source, and is available for Windows, Linux and MacOS operating systems. Furthermore, it has extensive support to expand the capabilities of the editor with packages, the equivalent of Visual Studio's extensions.

The motivation for developing a package for Atom is not to match or possibly even exceed the features offered by Dafny IDE [19]. We aim to offer a verification experience that enables productivity for more advanced users, while maintaining simplicity for beginners. Additionally, we want to make it easy for users to get started without the need of complicated installation steps or configuration. Atom provides a solid foundation to accomplish these goals. It offers rich text editing capabilities to give even the most advanced programmers the tools needed to write their code quickly, without overwhelming more casual programmers. Additionally, it has a large package ecosystem that makes it easy to develop, share and install those packages by providing a built-in package manager and central package repository [4].

5.4 Details and Features

Two separate packages have been developed to aid writing Dafny programs in Atom: `language-dafny` and `dafny-workbench`.

5.4.1 language-dafny

`language-dafny` is a so called grammar package, which adds support for the Dafny programming language to Atom by associating certain files, i.e. files with the `.dfy` file extension, with the Dafny grammar. This enables other packages to detect when a Dafny file is being opened in the editor. Furthermore, it provides syntax highlighting to files associated with the Dafny grammar, so programmers can visually distinguish between different grammar constructs.

[1] http://rise4fun.com/dafny
[2] https://atom.io/
[3] https://github.com/
[4] https://atom.io/packages

5.4.2 dafny-workbench

`dafny-workbench` provides the interaction between the text editor and the dafny command line interface. It invokes Dafny on the contents of a text editor once the changes because stable, i.e. after a certain time interval in which the contents do not change. By default this inactivity time interval is 300 miliseconds. This avoids needlessly invoking Dafny after a single character is changed, yet ensures the editing experience feels 'snappy' and aims to give the programmar the impression Dafny is continuously looking over their shoulder to give feedback at design-time of their program. If a user makes a change while Dafny is already running in the background, that instance of Dafny is then stopped and a new one is started on the modified buffer. It is important to note that in our case the complete contents of the text editor's buffer are passed to Dafny, with no caching. This is different and less efficient than the way the Dafny IDE (Dafny IDE)[19] opperates, since we have to re-verify all methods and functions, even if they were not changed. Nevertheless, we believe we can still provide a snappy experience: in most Dafny programs used in education the verifier finishes in the order of a couple seconds, often even faster.

Continuous Feedback

After a Dafny file is changed (and a certain time has passed in which no change occurred), Dafny will be invoked on the complete file, even if the file is not saved to disk. The status bar will show a spinner to indicate Dafny is indeed running, so the user is not left wondering whether or not something is happening in the background, see Figure 5.1. Furthermore, the status icon can be clicked at all times, either to stop a verification task if it is running or to restart a verification task when it is not. Dafny's output will be processed and provided to the user. How this happens depends on the result of the verification. In all cases a status icon is shown to visually indicate the result. If the programmer moves their mouse over the icon, a tooltip is shown with more information, for example how long the verification took and how many methods or functions were proven correct and incorrect, just like the Dafny's command line interface would.

An example of successful verification can be seen in Figure 5.2. A green icon with a check mark is shown in the status bar to indicate Dafny successfully verified the program correct.

An example of unsuccessful verification can be seen in Figure 5.3. A red icon with a question mark is shown in the status bar to indicate Dafny did not succeed in verifying the program correct. A list with the problems is shown, in this case finalization cannot not be proven. A programmar can click on the items in the list to jump to the relevant line in the program. Additionally, a red dot is shown in the gutter, i.e. the margin next to the text editor, to provide a visual indication in the Dafny program itself.

Another example of unsuccessful verification can be seen in Figure 5.4. In this case Dafny is unable to prove local correctness and interference-feedom. The issue is the `ensures` clause in line 15: `x == 2 || x == 2;`. The second parallel component is not locally correct, because our post-condition does not hold when we start with

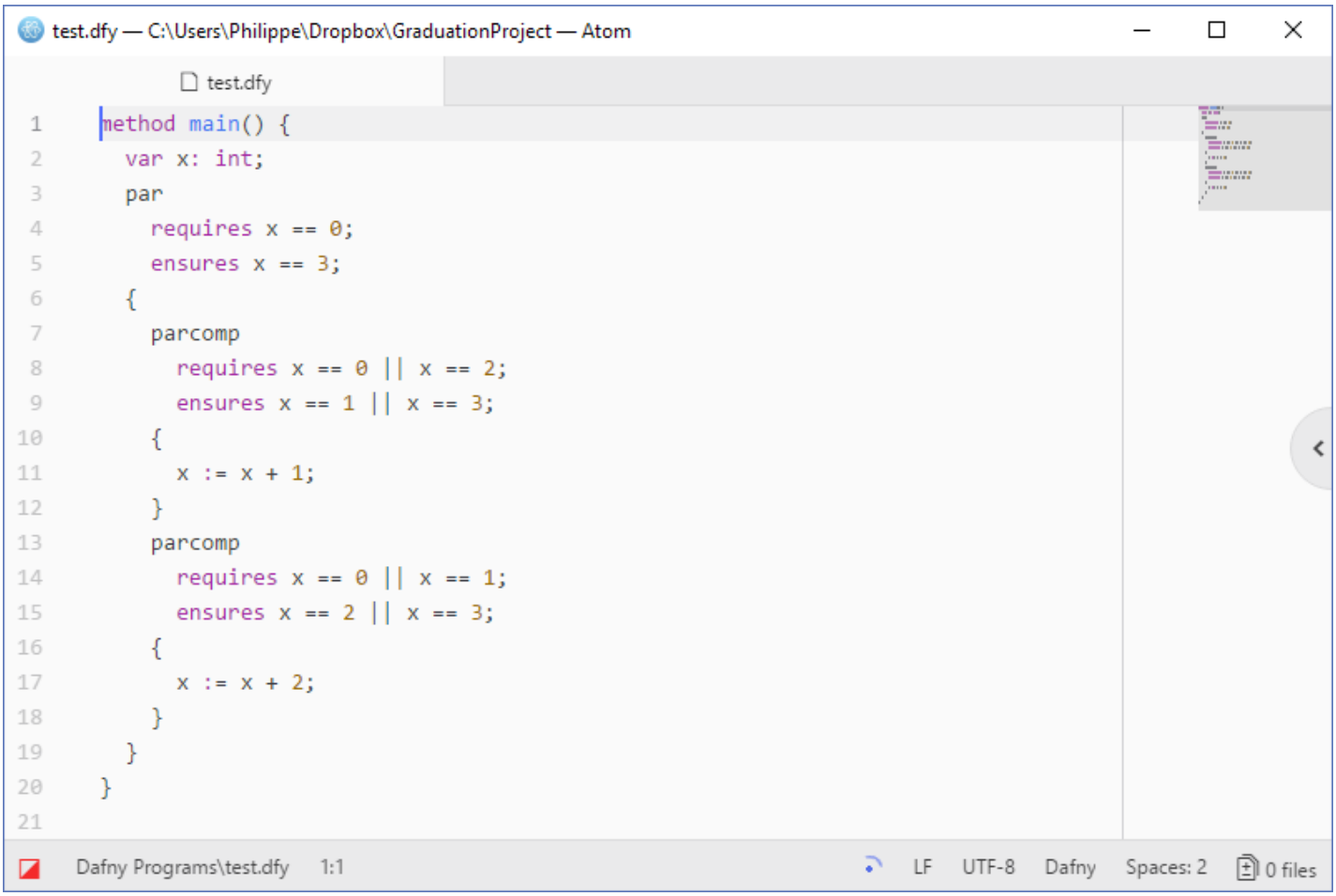


Figure 5.1: A pending verification

`x == 1`. The post-condition is also not interference-free, because the statement in line 11 interferes with it. If we make the necessary change and correct the `ensures` clause to `x == 2 || x == 3;` we obtain the program in Figure 5.2, which Dafny can verify correct.

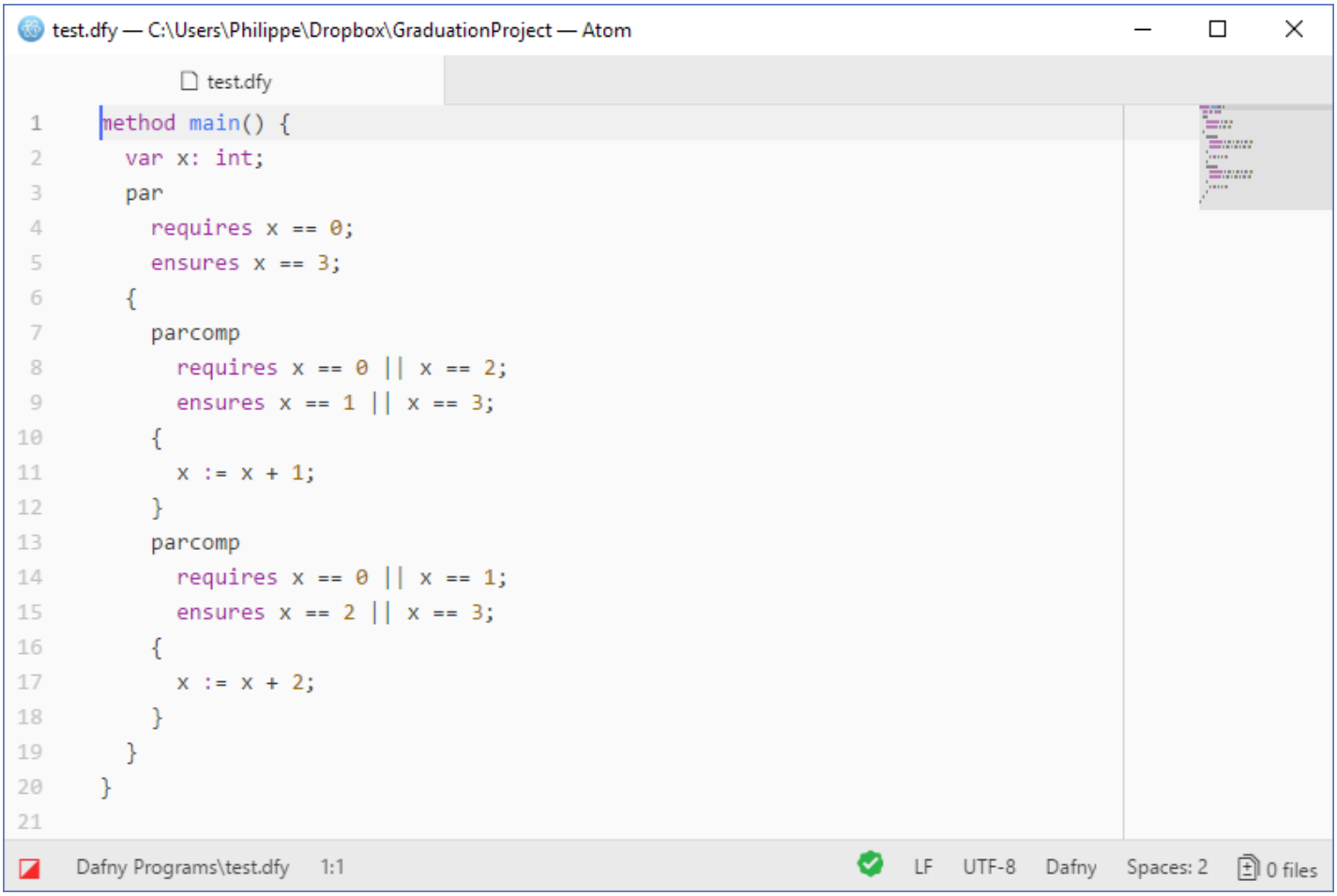


Figure 5.2: A successful verification

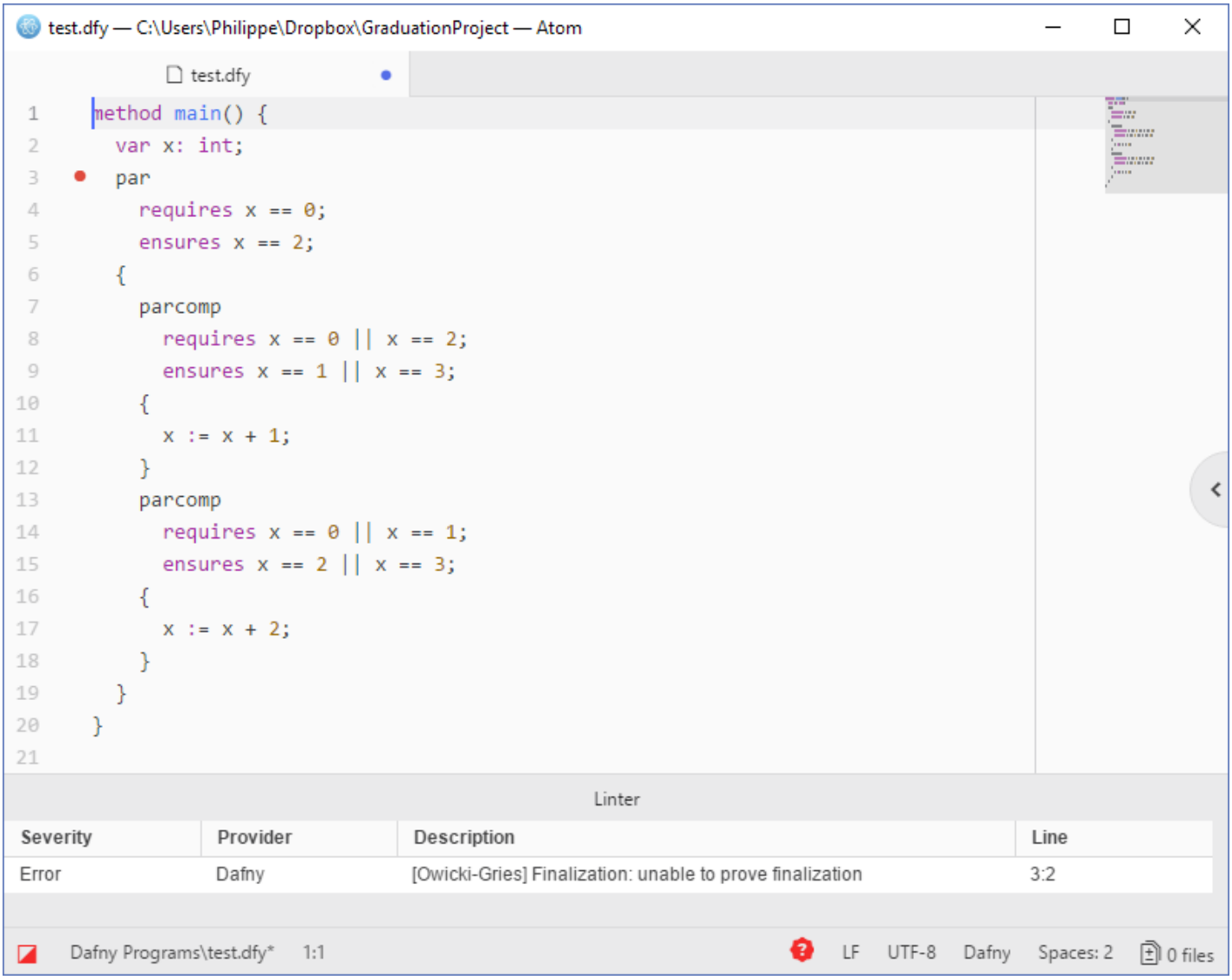


Figure 5.3: An unsuccessful verification: finalization cannot be proven

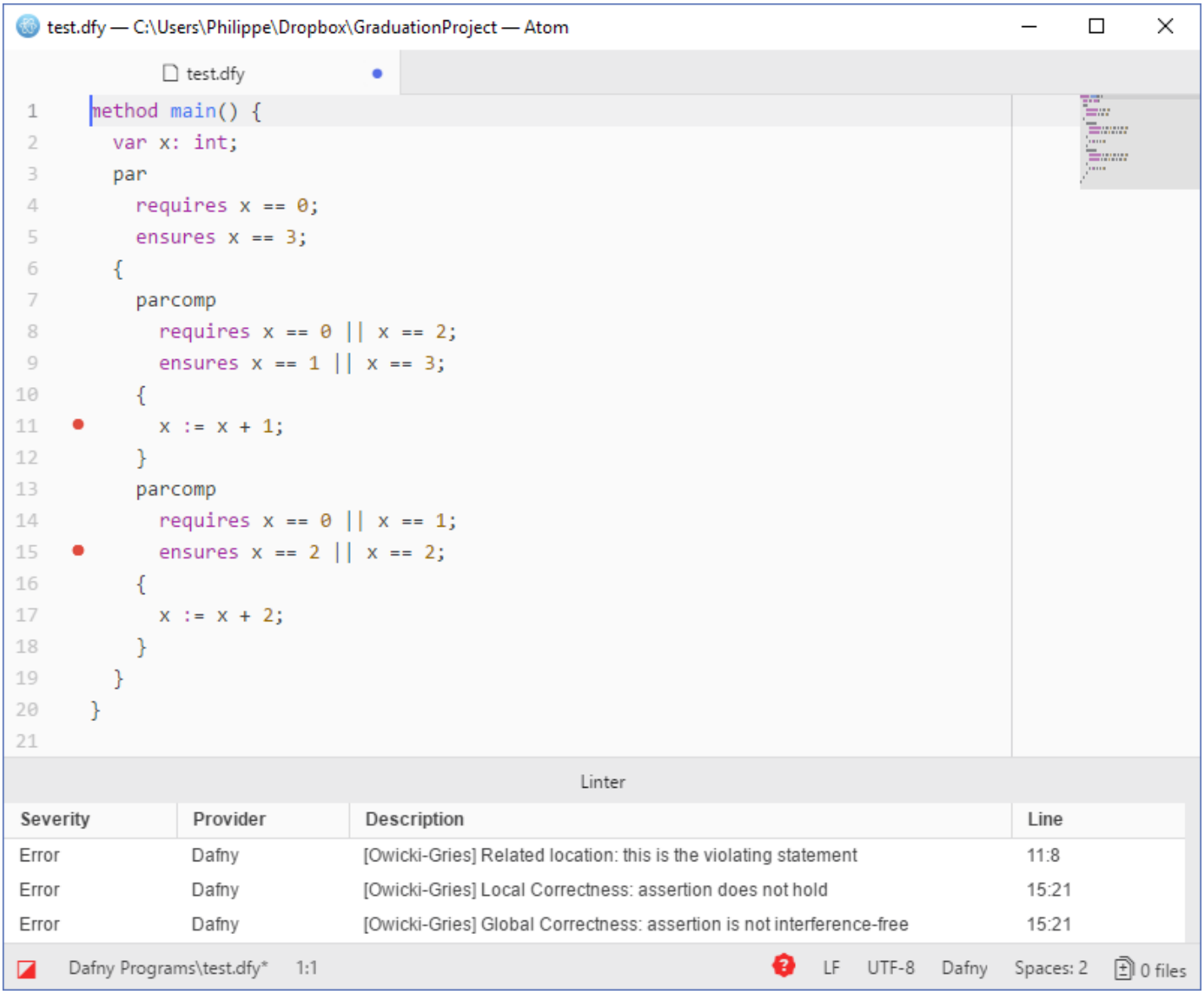


Figure 5.4: An unsuccessful verification: local correctness and interference-freedom cannot be proven

Part II

Implementation of Dafny Extension

Chapter 6

Language Extension: Concurrent Dafny Language

Dafny has no notion of concurrency, so the addition of Owicki-Gries verification requires an extension of the language. There are a few design criteria and restrictions we want to impose on the extension. The goal is to not only add Owicki-Gries verification, but also to make it easy for students to work with. As such, we aim to make the syntax close to what students would expect after studying the theory. Furthermore, it should fit within Dafny's current syntax and adhere to the style and patterns already in use. Additionally, we want the footprint to be as small as possible. This not only makes it easier for the user to express a concurrent program, but also makes it easier to maintain Dafny itself.

Several ideas have been considered, but have been dismissed, often because they were impractical or led to complications. In literature parallel composition of two program components is often notated by two pipe characters (`||`. This would translate to Dafny as the addition of a binary operator `||`. Unfortunately, this binary operator already exists in Dafny and denotes the logical `OR`, so we would have to take three pipe characters instead. The clear downside of this approach, besides a potentially confusing operator, is that program components consist of multiple statements and span more than a single line, which makes the code difficult to read. We could define our program components in a method and only allow method references in the parallel operator, but this means Dafny will try and verify our program components with its normal verification techniques. As shown in Section 7.3, this could lead to unexpected results.

We opted for an approach that lies closely to the way Dafny denotes methods and their pre- and postconditions, with some elements from the `calc` statement. We believe that anyone familiar with Dafny will immediately recognize the syntax, while it is also easy to make a mapping from the standard proof outline to the actual Dafny code.

6.1 Grammar

Dafny's grammar uses Extended BNF notation and Coco/R is used as the lexer and parser generator. A complete description of the original Dafny grammar can be found in the Dafny Reference Manual. [1] Full documentation can be found in the Coco/R Reference Manual. [2] A small summary to understand the grammar extension:

- | separates alternatives
- () groups alternatives
- {} denotes an iteration of zero or more times

In this document a production rule starts with an identifier and its definition follows on indented subsequent lines. The specification has been simplified by removing certain details relevant for the parser, but irrelevant to understand the grammar.

The formal specification of the additional production rules for the concurrency extension can be found in Listing 6.1.

```
ParStmt =
  "par" { ReqEnsSpec } "{" { ParCompLst } "}"

ReqEnsSpec =
  RequiresClause_ | EnsuresClause_

ParCompLst =
  { ParCompDecl }

ParCompDecl =
  "parcomp" { ReqEnsSpec } "{" { ( ParCompBody | OneStmt ) } "}"

ParCompBody =
  ParCompOneStmt

ParCompOneStmt =
  ( AssertStmt | UpdateStmt | WhileStmt )
```

Listing 6.1: Additional production rules for Concurrency Extension

The `ParStmt` production rule is added as an alternative to Dafny's `OneStmt` production rule. This implies we are allowed to define parallel programs inside the body of a `method`.

Parallel programs are always anonymous, we do not allow them to be named and referenced directly to maintain simplicity. If for some reason this is desired, one could wrap the parallel program inside a method and reference the method instead.

The production rules are straight forward. A parallel program defines its pre- and

[1] https://github.com/Microsoft/dafny/raw/master/Docs/DafnyRef/out/DafnyRef.pdf
[2] http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/Doc/UserManual.pdf

postconditions, as well as its parallel components. Each parallel component defines their pre- and postconditions, as well as the statements in their body.

Careful analysis of the grammar will reveal that a LL(1) conflict may occur. `( ParCompBody | OneStmt )` inside the `ParCompDecl` production is the offending part. The alternatives exist to only allow statements from `ParCompBody` inside parallel components, while still being able to parse any other statements to warn the user they are not allowed inside parallel components. Otherwise the user would only receive a syntax/parsing error, which could be confusing if it is perfectly valid Dafny outside a parallel component. The conflict arises because `ParCompOneStmt` is a subset of `OneStmt`. If a statement exists in both the parser does not know which alternative to choose. To make a decision the parser will choose the first one (from `ParCompOneStmt`) [3]. This is exactly what we want. To alert us of the conflict Coco/R will show a warning while generating the lexer and parser, which can be safely ignored. A solution to get rid of the warning and the LL(1) conflict is straight forward. We could create a `ParCompDisallowedOneStmt` prodution rule containing `OneStmt` minus `ParCompOneStmt`. However, this creates extra maintenance overhead, as any statement added to `OneStmt` also needs to be added to `ParCompDisallowedOneStmt`. It is convenient that any new statement added to Dafny will not be allowed inside a parallel component, as it likely needs additional verification rules for Owicki-Gries.

We do not allow assumption statements. These can be used in Dafny during proof construction to assume correctness of a certain expression, allowing a part of the proof to be completed. Assumptions are very helpful to speed up the creation of proofs, but unfortunately their meaning in parallel programs is not clear. An assume statement could imply that the assertion should be considered locally correct, but could also imply that the assertion should be considered interference-free, or perhaps both. None of the options are truly intuitive and there are scenarios conceivable that plead for any of the three options, so we leave out assumption statements instead.

6.2 Syntax

Consider the parallel program specification discussed in Section 2:

```
            { x == 0 }
[ x := x + 1 || x := x + 2 ]
            { x == 3 }
```

With the following standard proof outline:

```
                    { x == 0 }
{ x == 0 \/ x == 2 }    { x == 0 \/ x == 1 }
     x := x + 1              x := x + 2
{ x == 1 \/ x == 3 }    { x == 2 \/ x == 3 }
                    { x == 3 }
```

[3] Section 2.4.5 of Coco/R Reference Manual

The syntax to translate this standard proof outline into Dafny (with Extension) is as follows:

```
par
  requires x == 0;
  ensures x == 3;
{
  parcomp
    requires x == 0 || x == 2;
    ensures x == 1 || x == 3;
  {
    x := x + 1;
  }
  parcomp
    requires x == 0 || x == 1;
    ensures x == 2 || x == 3;
  {
    x := x + 2;
  }
}
```

A `var x: int;` prior to the `par` statement is needed to declare the variable, but to keep the translation simple and minimal it has been omitted so we can concentrate on the one-to-one translation of (standard) proof outline to Dafny code.

We define the parallel program with a `par` statement. The pre- and postcondition of the parallel program are defined in the corresponding `requires` and `ensures` clauses. Inside the body of the `par` statement we define the two parallel components with two `parcomp` statements.

Each `parcomp` statement has its pre- and postconditions defined with a `requires` clause and `ensures` clause, respectively. These pre- and postconditions do not come directly from the program specification, but are created by the user, similarly to creating a (standard) proof outline by hand. Inside the body of the `parcomp` statement we define the statements of each parallel component, in this case a simple assignment for each one.

Dafny can now verify this proof outline and if it is indeed correct, we may conclude it is a standard proof outline.

6.3 Atomicity

We go from a sequential language to a concurrent language, so we need to introduce a concept of atomicity, in which we will follow the semantics as defined by [1]. We do not introduce separate syntax to define atomic regions, but define them implicitly instead. There are consequences to this approach and it could lead to unexpected results, as we will see in Section 7.3.

We define everything between two assertions (with no other assertions in between) inside a `parcomp` as one atomic region, which means that all statements between two

assertions are evaluated or executed as one atomic action. In particular, boolean expressions, assignments and multi-assignments are evaluated or executed as if they are one atomic action. This assumption will rarely be garantueed by most programming languages or hardware, which typically garantuee only exclusive access to reading from or writing to a shared variable.

We can circumvent this problem by making sure every atomic region, i.e. everything between two assertions, contains at most one shared variable access. Consider the assignment $x := y$, where x and y are shared variables. In traditional programming languages (and hardware) this would require a read of y and a write of x, which will happen in one atomic step even though that is exactly what we model with our syntax. The solution is to create an additional variable, for example *temp*, that holds the value of y: $temp := y$. In the next step, we assign this value of *temp* to x: $x := temp$. Provided we place the correct assertions in between those two assignments, we have modelled an assignment by splitting it into statements that only read or write to one shared variable.

For educational purposes this approach does not hinder us. We are interested in the verification of parallel programs and often the coarse granularity is fine to illustrate the principles. If needed, we can always achieve a finer granularity by splitting each statement into multiple statements as shown above.

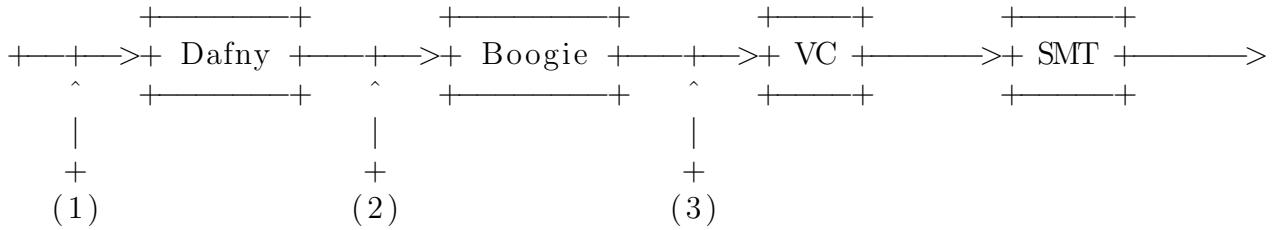
Chapter 7

Translation of Concurrent Dafny to Boogie

The Owicki-Gries method of verifying parallel programs requires us to show initialization, finalization, local correctness and interference-freedom. Each of the four requirements have their own proof obligations, but where in the verification step do we generate the proof obligations associated with the method? To answer this question, we must first understand what happens under the hood when a Dafny program is verified.

7.1 Generating Proof Obligations

The Dafny program is translated into an intermediate verification language (IVL), i.e. a Boogie program. Then, the Boogie program is translated into a set of first-order verification conditions (VC). Finally, these verification conditions are passed to a theorem prover, i.e. a satisfiability-modulo-theories (SMT) solver, e.g. Z3. A visual representation of the Dafny pipeline is given in Figure 7.1.



This presents us with various opportunities at which we can generate the proof obligations for Owicki-Gries, each indicated with a number. We will briefly discuss each of the options and explain our final choice.

(1)

We could generate Dafny code that includes all the proof obligations for Owicki-Gries, e.g. by using a preprocessor that transpiles Dafny code augmented with comments indicating parallel components and their pre- and postconditions.

- Pros:
  - Simple to implement.
- Cons:
  - Dafny has no concept of Owicki-Gries, no design-time feedback.
  - Dafny proves the generated code, so difficult to relate violations found by the prover back to the original (augmented) Dafny code.
  - Requires a deep understanding of Owicki-Gries to write and debug proof obligations.

This approach has been implemented as an experiment and proof of concept, see Section 3.1.

(2)

We could extend Dafny with syntax to define program components and generate the necessary proof obligations in the transformation step to Boogie.

- Pros:
  - Well-defined syntax.
  - Dafny has a concept Owicki-Gries, design-time feedback.
- Cons:
  - Boogie has no concept of Owicki-Gries.

This approach is a logical next step after implementing (1). We can make use of Dafny's machinery and embed Owicki-Gries support within Dafny itself. This allows us to fully integrate Owicki-Gries, including a clear syntax and user feedback, something that was not possible with approach (1). This is the the approach we choose for our implementation.

(3)

Alternatively, we could extend Dafny and Boogie with syntax to define program components and generate the necessary proof obligations in the transformation step to first-order verification conditions.

- Pros:
  - Well-defined syntax.
  - Dafny and Boogie have a concept of Owicki-Gries, fully integrated into the prover system.
  - Other verifiers using Boogie can lean on its Owicki-Gries support to simplify the implementation of their own.
- Cons:

– Likely difficult to implement and maintain, need to add support to both Dafny and Boogie.

This approach opens up the possibility for other verifiers to make use of Boogie's Owicki-Gries implementation. The details of Owicki-Gries would only need to be implemented once at the Boogie level, and all other verifiers can 'borrow' from it and would only need to add Owicki-Gries support at a higher level. All proof obligations could be taken care of by Boogie, while for example Dafny only worries about the syntax and delivering feedback to the user.

We decided against this option because of the extra complexity in the implementation and the relatively low importance of having Owicki-Gries support in Boogie itself. Not all verifiers that make use of Boogie have a need for Owicki-Gries. Chalice, for example, is specifically tailored towards proving concurrent programs and as such, it has no use for an Owicki-Gries extension.

7.2 Approach

Consider a Dafny statement `S1`, which is translated to Boogie with a series of statements `S1'` and a Dafny statement `S2`, which is translated to Boogie with a series of statements `S2'`. Both `S1` and `S2` are contained in the same method in Dafny and are translated to the same procedure in Boogie. This implies that to proof `S2`, Dafny may infer information that was valid in `S1` (provided no other statements prevent it) to prove statement `S2`, which in practise means that Boogie uses `S1'` to prove `S2'`.

Normally, all statements in Dafny are sequential, so this approach is valid. However, when we add Owicki-Gries statements, we have a parallel situation. If `S1` is a statement of the first program component of a parallel statement, and `S2` is a statement of the second program component of a parallel statement, and we translate it like Dafny normally would, we have a problem. Even if we produce Hoare triples for `S1` and `S2`, those Hoare triples should be proven in isolation, without allowing information from `S1` to influence the proof of `S2`. As such, we cannot translate an Owicki-Gries statement to a single procedure in Boogie, like Dafny would translate any other statement.

To solve this problem, we translate each Hoare triple that follows from the Owicki-Gries statement to a different procedure in Boogie to ensure isolation.

Dafny		Boogie
		...
...		S1'
S1	–>	S1'
...		S1'
		...
		...
...		S2'
S2	–>	S2'

```
...          S2'
             ...
```

Consider the following program:

```
method Main()
{
  var x: int;
  var A: int;
  var B: int;
  par
    requires A == B == 0 || (A != B && A != -B);
    requires x == 0;
    ensures x == A + B;
  {
    parcomp
      requires x == 0 || x == B;
      ensures x == A || x == A + B;
    {
      x := x + A;
    }
    parcomp
      requires x == 0 || x == A;
      ensures x == B || x == A + B;
    {
      x := x + B;
    }
  }
}
```

To proof that the `par` statement is indeed correct according to the theory of Owicki-Gries, we need to proof the following:

- initialization
- finalization
- local correctness of both `parcomp` statements
- interference-freedom of both `parcomp` statements

Let us look at the initialization. We are required to show that the following Hoare triple is indeed valid:

```
{ A == B == 0 || (A != B && A != -B) }
{ x == 0 }
// empty command
{ x == 0 || x == B }
{ x == 0 || x == A }
```

We can do this by generating a Boogie `procedure` that takes the relevant variables as in-parameters, with our pre-condition as `requires` clauses and our post-condition

as `ensures` clauses. The corresponding Boogie `implementation` can have an empty body, but cannot be omitted! Omission would mean that Boogie simply skips verification of the `procedure` and assumes it is correct, which is not what we want.

```
procedure Initialization();
  requires (A#0 == B#0 && B#0 == LitInt(0)) || (A#0 != B#0 && A#0 != 0 -
  requires x#0 == LitInt(0);
  ensures x#0 == LitInt(0) || x#0 == B#0;
  ensures x#0 == LitInt(0) || x#0 == A#0;

implementation Initialization()
{
}
```

Of course Boogie will not not compile this piece of code, because we never declare the identifiers. A simple fix would be to add them as in-parameters, so that we may use them in `requires` and `ensures` clauses, as well as statements in the `implementation` part. Like so:

```
procedure Initialization(x#0: int, A#0: int, B#0: int);
  requires (A#0 == B#0 && B#0 == LitInt(0)) || (A#0 != B#0 && A#0 != 0 -
  requires x#0 == LitInt(0);
  ensures x#0 == LitInt(0) || x#0 == B#0;
  ensures x#0 == LitInt(0) || x#0 == A#0;

implementation Initialization(x#0: int, A#0: int, B#0: int)
{
}
```

This approach works fine for initialization and finalization, but breaks down for local correctness and interference-freedom, where the Hoare triples actually have a command. Consider local correctness of the first `parcomp` statement:

```
{ x == 0 || x == B }
x := x + A
{ x == A || x == A + B }
```

Which would lead to the following Boogie `procedure` and `implementation`:

```
procedure LocalCorrectnessParcompOne(x#0: int, A#0: int, B#0: int);
  requires x#0 == LitInt(0) || x#0 == B#0;
  ensures x#0 == A#0 || x#0 == A#0 + B#0;

implementation LocalCorrectnessParcompOne(x#0: int, A#0: int, B#0: int)
{
  x#0 := x#0 + A#0;
}
```

Unfortunately, this is not valid Boogie, because the parameters are immutable: we are not allowed to modify them, only use as is.

We can solve this problem in at least the following two ways:

- Use shadow variables as parameter
- Use global variables

7.2.1 Shadow Variables

The idea of shadow variables comes down to the following. Note that `x'` is a valid identifier for a variable in Boogie.

```
procedure LocalCorrectnessParcompOne(x#0': int, A#0': int, B#0': int) retu
  requires x#0' == LitInt(0) || x#0' == B#0';
  ensures x#0 == A#0 || x#0 == A#0 + B#0;

implementation LocalCorrectnessParcompOne(x#0': int, A#0': int, B#0': int)
{
  x#0 := x#0';
  A#0 := A#0';
  B#0 := B#0';
  x#0 := x#0 + A#0;
}
```

For each variable `x` we create a new shadow variable `x'`. We use `x'` as in-parameter and in the `requires` clauses. In the implementation, we assign the value of `x'` to `x`. We add each variable `x` in the return statement so that we can make use of it in the `ensures` clause.

When trying to implement this technique it turns out that it is nearly impossible to accomplish. Renaming the in-parameters to their primed versions is straightforward. The difficulty lies in converting the `requires` clause: it can contain arbitrary expressions (e.g. `x == 0`, `myMethod(x)`, etc.). There is no easy way in Dafny to iterate over all the identifiers used in those expressions. With signicant engineering effort it would be possible to replace the identifiers with their primed counter-part, but it is hard to justify a large modification to the internals of Dafny. It would create extra overhead on maintaining the core non-concurrent part of Dafny and we do not want to disturb the stability of the internals of Dafny unless absolutely necessary.

7.2.2 Global Variables

An alternative is to simply use global variables. The reason for trying to avoid this is simple: we do not wish to pollute the global namespace if it is not strictly necessary. Unfortunately, avoiding it proved to be difficult and this is the solution we implemented.

In Boogie we may only declare a top level variable once or else we receive the following message: `Error: more than one declaration of variable name`. The solution is straight forward, we only add a variable when it does not already exist.

7.2.3 Limitations

Conditions on constants need to be added by the user to each `par` statement and its `parcomp` statements. We believe the use-case for imposing restrictions on constants are limited and cannot justify the work involved in creating a new mechanism to provide parallel program wide conditions on variables. An idiomatic approach to circumvent this limitation is by capturing the conditions in a function and use the function in `requires`, `ensures`, or `assert` statements. The conditions themselves will be defined only once in one place, but the function still needs to be added everywhere the conditions are needed.

7.3 Local Correctness

We have already seen that local correctness is essentially proving each program component correct, in other words just a sequential proof. The Owicki-Gries theory describes how to do this: we simply proof every Hoare-triple in the program component correct. However, it is easy to make the observation that this might not be needed, because Dafny already has all the machinery to show correctness of sequential programs. Thus, the logical next step would be to let Dafny proof local correctness of each program component using its normal set of rules, heuristics, and so on.

Unfortunately, this approach is flawed and will lead to unexpected results. To illustrate the problem, we will give an example of a parallel program that is clearly incorrect, yet a proof that uses vanilla Dafny to show local correctness will result in the inability to detect the problem in the interference-freedom check and falsely show correctness.

```
                { x == 0 }
{ x == 0 }                      { true }
 x := 1;                      if (x == 1)
                ||                x := 2;
 y := 2x;                     else
{ y == 2 }                        x := 0;
                               { true }
                { y == 2 }
```

Listing 7.1: Incorrect: missing assertion between two statements of the left program component

The parallel program in Listing 7.1 is incorrect. If the right program component executes its `if`-statement in between the two assignments of the left program component, the value of x will be set to 0, which means that y in the next line will be assigned with 0 as well. Consequently, the parallel program will terminate with a value of $y == 0$, which clearly contradicts the parallel program's post-condition of $y == 2$.

Initialization and finalization are trivially satisfied, i.e. $x == 0 \implies x == 0 \wedge true$ and $y == 2 \wedge true \implies y == 2$ both hold.

When we look at both program components in isolation we can easily show that each of them is locally correct, something that Dafny would also be able to show. The absence of an assertion between the two assignments of the left program component is not a problem, Dafny will infer from the context the information it needs to show correctness.

Unfortunately, global correctness will also be satisfied. The right program component has two `true` assertions, which are trivially interference-free. The left program component has a $y == 0$ assertion, which is also trivially interference-free, since y does not occur in the other program component. Lastly, the remaining $x == 0$ assertion is also interference-free.

According to the theory of Owicki-Gries we have shown correctness of initialization, finalization, local and global correctness and we could (erroneously) conclude that the parallel program in Listing 7.1 is proven correct. However, we already know that it is not correct.

```
                { x == 0 }
{ x == 0 }                      { true }
 x := 1;                       if (x == 1)
{ x == 1 }        ||                x := 2;
 y := 2x;                      else
{ y == 2 }                         x := 0;
                               { true }
                { y == 2 }
```

Listing 7.2: Correct: assertion between two statements of the left program component

The problem lies in the missing assertion between the two assignment statements of the left program component. Consider the parallel program in Listing 7.2, which is identical to the one in Listing 7.1 except for the assertion between the two assignment statements. Initialization, finalization and local correctness are still satisfied. This time, however, global correctness is *not* satisfied anymore, because $x == 1$ interferes with the if-statement of the right program component.

The important observation is the following: we cannot use vanilla Dafny to prove local correctness of the program components, because Dafny allows us to leave out assertions that could be needed to show there is an interference issue. These missing insertions are automatically inferred, but we do not have direct access to them from within Dafny and as such we cannot automatically add them to the interference-freedom checks. This problem is related to our implicit definition of atomicity. By considering everything between two assertions as one atomic statement the issue with Listing 7.1 can no longer occur, but nevertheless we do not use vanilla Dafny and stay close to the Owicki-Gries theory to show local correctness. This has the added benefit that the system remains consistent should we want to introduce syntax to define an atomic statement at a later point in time.

7.3.1 Proof Rules

To show that a proof outline is indeed a standard proof outline, we can use the axioms and rules corresponding with the proof system introduced by [1] to prove partial correctness of while programs.

Different type of statements call for different proof rules. We will show the required implications or Hoare triples for each statement we allow inside a parallel component.

Update Statement

Consider an update statement S with precondition P and postcondition Q. We need to show validity of the following Hoare triple:

```
{ P }
  S
{ Q }
```

While Loop

We only support partial correctness of While-loops inside parallel statements, i.e. we do not support proving termination.

Consider a While-loop of the form:

```
{ P }
while (G)
    invariant I
{
    { A }
    S
}
{ Q }
```

We need to show validity of the following implications:

- $P \implies I$
- $I \wedge G \implies A$
- $I \wedge \neg G \implies Q$

Lastly, we need to show validity of the following Hoare triple:

```
{ I /\ G}
    S
  { I }
```

7.4 Interference-Freedom

7.4.1 Proof Rules

Update Statement

Consider an update statement S with precondition P and postcondition Q, which leads to the following Hoare triple:

```
{ P }
  S
{ Q }
```

We need to show Interference-Freedom of the following assertions:

- P
- Q

While Loop

Consider a While-loop of the form:

```
{ P }
while (G)
    invariant I
{
    { A }
    S
}
{ Q }
```

We need to show Interference-Freedom of the following assertions:

- P
- I
- A
- Q

7.5 Soundness

To achieve soundness, we require the following two assumptions about our concurrency model:

- atomicity of elementary statements
- sequential consistency (no change in execution order of elementary statements)

Assuming sequential consistency is crucial. If we cannot rely on the execution order (e.g. because of reordering due to compiler or processor optimizations) of the statements within each program component the Owicki-Gries method breaks down completely, because suddenly assertions are no longer valid. Furthermore, we assume atomicity of elementary statements to make our life easier. As shown in Section 6.3 this assumption can be done without loss of generality, since a finer granularity can be achieved by splitting each statement into separate statements that only read or write to one shared variable.

We generate proof obligations based on the proof rules presented by [1]. These proof rules are proven sound w.r.t. the semantics in [1]. So, assuming the soundness of Dafny w.r.t. this semantics, our verificatons are also sound w.r.t. this semantics. Complications arise that are addressed in this thesis.

These proof rules require a standard proof outline. According to Section 8.4 of [1], a proof outline for partial correctness is standard if every statement of the program is preceded by exactly one assertion (and there are no assertions that are not such a preceding assertion) and in particular, there are no assertions within atomic regions. We obtain this by design, since we defined our atomic regions in Section 6.3 to span from one assertion to the next, without any assertions in between.

The sequential part of Dafny has similar semantics to that of [1] when it comes to `update`-statements and `while`-loops. The parallel part of Dafny, i.e. the way in which we handle the different statements, has similar semantics to that of [1] by design, because that is the base we use for generating our proof obligations.

We just argued that soundness is maintained while generating the proof obligations for a parallel program. These proof obligations are given to Boogie, which translates them into a set of first-order verification conditions to be passed to Z3. We may assume soundness of Boogie and as such, soundness of the translation to Z3 and the theorem prover itself. Soundness of sequential programs is not endangered, because our extension does not touch the translation from Dafny to Boogie for the sequential part.

Dafny is also a compiler and is able to compile verified programs to C#. To maintain soundness for the compilation of Dafny programs that include parallel statements, it is essential that the semantics of these compiled statements are equal to those of [1]. This is not part of this thesis, but is mentioned as Future Work in Section 9.1.

Owicki-Gries is relatively complete. Dafny's assertion language has the same expressiveness as that of [1], including ghost variables. Therefore we also maintain relative completeness: every semantically valid (i.e. true) formula can be proven, modulo the incompleteness for arithmetic.

Part III

Conclusion

Chapter 8

Case Study

Some concurrent programs can be easily proven with Owicki-Gries, but even if the steps in the proof are simple, there is still a high risk of making mistakes. Most proof obligations are not difficult but rather tedious if one needs to proof them by hand. It is easy to fall victim to the temptation of being sloppy, for example carelessly going through proof obligations that look similar to several that have already been completed, only to find out later (or not at all!) that it actually leads to an incorrect proof.

Dafny forces you to make the correct, possibly difficult to find, assertions and write them down precisely. Dafny will then calculate if those assertions are correct, so the user need not write down the small steps or proofs that lead to the correct assertion.

Clearly, this has the advantage that tedious bookkeeping is no longer an obligation for the user. However, it is important to note that all assertions still need to be mentioned explicitly, no assertions are filled in automatically.

To illustrate the need to be meticulous and extremely precise, we will give an example of a concurrent program that seems simple, but needs a careful analysis in order to find all the criteria on its input before correctness can be established. This clearly argues for a tool to check all the tedious steps. Section 5.4 shows that the tooling developed during this thesis for our Owicki-Gries implementation in Dafny will immediately point that there is a problem if we are not careful in defining our conditions on the constants.

Let us consider the following parallel program specification, with A and B integer constants:

$$\{x = 0\}\ [x := x + A \parallel x := x + B]\ \{x = A + B\} \tag{8.1}$$

We will show partial correctness, without the use of auxiliary variables. To accomplish this, A and B should satisfy certain conditions, which we will derive along the way.

8.0.1 Manual Proof

Consider the following proof outlines, where we have weakened the pre- and post-conditions of each component of the the parallel composition.

$$\{x = 0 \vee x = B\}\ x := x + A\ \{x = A \vee x = A + B\} \tag{8.2}$$

$$\{x = 0 \vee x = A\}\ x := x + B\ \{x = B \vee x = A + B\} \tag{8.3}$$

We will use 8.2 and 8.3 to prove partial correctness of 8.1. Owicki-Gries requires us to establish local correctness and interference freedom, as well as initialization and finalization. [1]

Local Correctness

First consider 8.2. By the assignment axiom, we have that:

$$\{x + A = A \vee x + A = A + B\}\ x := x + A\ \{x = A \vee x = A + B\} \tag{8.4}$$

Also note that we have:

$$x = 0 \vee x = B \rightarrow x + A = A \vee x + A = A + B \tag{8.5}$$

as this is equivalent to

$$x = 0 \vee x = B \rightarrow x = 0 \vee x = B$$

which is trivially true.

Furthermore, we trivially have

$$x = A \vee x = A + B \rightarrow x = A \vee x = A + B \tag{8.6}$$

Hence from 8.4, 8.5, 8.6 and the consequence rule follow that 8.2 is a correct proof outline.

Now consider 8.3. By the assignment axiom, we have that:

$$\{x + B = B \vee x + B = A + B\}\ x := x + B\ \{x = B \vee x = A + B\} \tag{8.7}$$

Also note that we have:

$$x = 0 \vee x = A \rightarrow x + B = B \vee x + B = A + B \tag{8.8}$$

[1] The full manual proof has been written in a slightly different form as a solution to an assignment for the course Program Verification Techniques. The original proof was done in collaboration between Philippe Denissen and Roel Jacobs.

as this is equivalent to

$$x = 0 \vee x = A \rightarrow x = 0 \vee x = A$$

which is trivially true.

Furthermore, we trivially have

$$x = B \vee x = A + B \rightarrow x = B \vee x = A + B \tag{8.9}$$

Hence from 8.7, 8.8, 8.9 and the consequence rule follow that 8.3 is a correct proof outline.

Interference-Freedom

In order to establish interference-freedom, we have to prove that $x := x + A$ does not interfere with the pre and post condition of $x := x + B$ and vice versa. To do this we have to prove that the following proof-outlines are correct

1. $$\{(x = 0 \vee x = B) \wedge (x = 0 \vee x = A)\}\ x := x + A\ \{(x = 0 \vee x = A)\} \tag{8.10}$$

2. $$\{(x = 0 \vee x = B) \wedge (x = B \vee x = A + B)\}\ x := x + A\ \{(x = B \vee x = A + B)\} \tag{8.11}$$

3. $$\{(x = 0 \vee x = A) \wedge (x = 0 \vee x = B)\}\ x := x + B\ \{(x = 0 \vee x = B)\} \tag{8.12}$$

4. $$\{(x = 0 \vee x = A) \wedge (x = A \vee x = A + B)\}\ x := x + B\ \{(x = A \vee x = A + B)\} \tag{8.13}$$

1. Consider 8.10. By the assignment axiom, we have

   $$\{x + A = 0 \vee x + A = A\}\ x := x + A\ \{(x = 0 \vee x = A)\} \tag{8.14}$$

   We will now show that

   $$(x = 0 \vee x = B) \wedge (x = 0 \vee x = A) \rightarrow (x + A = 0 \vee x + A = A) \tag{8.15}$$

   By distributivity and integer mathematics, 8.15 is equivalent to

   $$(x = 0) \vee (x = A \wedge x = B) \rightarrow (x + A = 0 \vee x = 0)$$

   This is only true in two cases

- $(A \neq B)$. Now $(x = 0) \vee (x = A \wedge x = B) \equiv (x = 0) \vee \text{FALSE} \equiv (x = 0)$, hence the implication holds by weakening.
- $(A = B = 0)$. Now $(x = 0) \vee (x = A \wedge x = B) \equiv (x = 0) \vee (x = 0 \wedge x = 0) \equiv (x = 0)$, hence the implication holds by weakening.

So we need the condition that $(A \neq B \vee A = B = 0)$ in order to prove partial correctness. So if we assume this condition is met, the implication is true.

Now from 8.14, 8.15 and the consequence rule we can conclude that 8.10 is a correct proof outline.

2. Consider 8.11. By the assignment axiom, we have

$$\{x + A = B \vee x + A = A + B\}\ x := x + A\ \{(x = B \vee x = A + B)\} \quad (8.16)$$

We will now show that

$$(x = 0 \vee x = B) \wedge (x = B \vee x = A + B) \rightarrow (x + A = B \vee x + A = A + B) \quad (8.17)$$

By distributivity and integer mathematics, 8.17 is equivalent to

$$(x = B) \vee (x = 0 \wedge x = A + B) \rightarrow (x = B - A \vee x = B)$$

This is only true in two cases

- $(A \neq -B)$. Now $(x = B) \vee (x = 0 \wedge x = A + B) \equiv (x = B) \vee \text{FALSE} \equiv (x = B)$, hence the implication holds by weakening.
- $(A = B = 0)$. Now

  $(x = B) \vee (x = 0 \wedge x = A + B) \equiv (x = B) \vee (x = 0 \wedge x = 0 + 0) \equiv x = 0$

  Also $(x = B - A \vee x = B) \equiv (x = 0 - 0 \vee x = 0) \equiv x = 0$. Hence the implication holds trivially.

So we need the condition that $(A \neq -B \vee A = B = 0)$ in order to prove partial correctness. So if we assume this condition is met, the implication is true.

Now from 8.16, 8.17 and the consequence rule we can conclude that 8.11 is a correct proof outline.

3. The proof for 8.12 is similar to (1.) with the roles of A and B interchanged.
4. The proof for 8.13 is similar to (2.) with the roles of A and B interchanged.

We have now established that 8.2 and 8.3 are locally correct and interference free. Hence we can conclude that

$$\begin{array}{r} \{(x = 0 \vee x = B) \wedge (x = 0 \vee x = A)\} \\ [x := x + A \parallel x := x + B] \\ \{(x = A \vee x = A + B) \wedge (x = B \vee x = A + B)\} \end{array} \quad (8.18)$$

is a correct proof outline, provided that $(A \neq \pm B \vee A = B = 0)$.

Conclusion

We now have to relate the result of 8.18 back to the original specification 8.1. We will show that the following two implications hold:

$$(x = 0) \rightarrow (x = 0 \vee x = B) \wedge (x = 0 \vee x = A) \tag{8.19}$$

$$(x = A \vee x = A + B) \wedge (x = B \vee x = A + B) \rightarrow (x = A + B) \tag{8.20}$$

First consider 8.19. Since

$$(x = 0 \vee x = B) \wedge (x = 0 \vee x = A)$$

is equivalent to

$$x = 0 \vee (x = B \wedge x = A)$$

we get the desired result by weakening.

Now consider 8.20. We have that

$$(x = A \vee x = A + B) \wedge (x = B \vee x = A + B)$$

is equivalent to

$$x = A + B \vee (x = B \wedge x = A)$$

Consider our assumption $(A \neq \pm B \vee A = B = 0)$.

- if $A \neq \pm B$, then $x = A+B \vee (x = B \wedge x = A) \equiv x = A+B \vee \text{FALSE} \equiv x = A+B$. Hence 8.20 holds trivially.
- if $A = B = 0$, then $x = A + B \vee (x = B \wedge x = A) \equiv x = 0 + 0 \vee (x = 0 \wedge x = 0) \equiv x = 0$. Also $x = A + B \equiv x = 0 + 0 \equiv x = 0$. Hence the 8.20 holds trivially.

From 8.18,8.19, 8.20 and the consequence rule, we can now conclude that the original specification 8.1 is correct, under the assumption $(A \neq \pm B \vee A = B = 0)$.

8.0.2 Dafny (with Extension) Proof

```
method Main()
{
  var x: int;
  var A: int;
  var B: int;
  par
    requires A == B == 0 || (A != B && A != -B);
    requires x == 0;
    ensures x == A + B;
    ensures A == B == 0 || (A != B && A != -B);
```

```
  {
    parcomp
      requires A == B == 0 || (A != B && A != -B);
      requires x == 0 || x == B;
      ensures x == A || x == A + B;
      ensures A == B == 0 || (A != B && A != -B);
    {
      x := x + A;
    }
    parcomp
      requires A == B == 0 || (A != B && A != -B);
      requires x == 0 || x == A;
      ensures x == B || x == A + B;
      ensures A == B == 0 || (A != B && A != -B);
    {
      x := x + B;
    }
  }
}
```

Listing 8.1: Dafny (with Extension) solution to 8.1

Listing 8.1 shows the Dafny (with Extension) solution to the parallel program specification in 8.1. Since Dafny does not support constants, the assumption we found in the manual proof, i.e. $A \neq \pm B \vee A = B = 0$, (see also Section 8.0.1) has to be added as a pre- and postcondition to the parallel program and all of its program components. An alternative would be to define the assumption only once in a predicate and add this predicate to the pre- and postconditions of the parallel program and all of its program components. Note that if we leave out the assumption, Dafny (with Extension) will no longer be able to verify our program as correct. Similar to the manual proof, we need to find this assumption ourselves. Dafny (with Extension) will then do all the calculations required to show program correctness by proving initialization, finalization, local correctness and global correctness.

Chapter 9

Future Work

9.1 Compilation

Dafny is not only a language and a verification tool, it is also a compiler. Programs written and verified in Dafny can be compiled to C#. C# (pronounced "C sharp") is a programming language that is designed for building a variety of applications that run on the .NET Framework [1]. Our implementation of Owicki-Gries in Dafny does not support compilation, but such a feature could be desirable. Given the extra complexity of writing concurrent programs, being able to execute a program that has been verified as correct is clearly beneficial.

There are several obstacles that prevent a straight-forward approach to adding compilation support to our Owicki-Gries implementation.

9.1.1 Atomicity

Our implementation does not have a separate syntax for atomic regions. Instead, they are implicitly defined by surrounding the atomic statements with assertions, see also Section 6.3. This implies a very high granularity and in the compilation step special care is required to ensure the target language has support for the same coarse-grained parallelism. The C# specification mentions the following about atomicity of variable references [2]:

> Reads and writes of the following data types are atomic: `bool`, `char`, `byte`, `sbyte`, `short`, `ushort`, `uint`, `int`, `float`, and reference types. In addition, reads and writes of enum types with an underlying type in the previous list are also atomic. Reads and writes of other types, including `long`, `ulong`, `double`, and `decimal`, as well as user-defined types, are not guaranteed to be atomic. Aside from the library functions designed for that purpose, there is no guarantee of atomic read-modify-write, such as in the case of increment or decrement.

[1] https://docs.microsoft.com/en-us/dotnet/csharp/csharp
[2] https://github.com/dotnet/csharplang/blob/master/spec/variables.md

The garantuees given by the specification are not sufficient for our needs, so a logical next step would be to employ a locking mechanism, like mutexes or semaphores. C# provides several methods for achieving thread synchronization: the `lock` keyword, monitors, event and wait handlers, a mutex object, and the interlocked class. [3]. One of our main concerns is efficiency. All these different mechanisms come with a certain overhead and we will pay the price for every atomic statement in every program component of our parallel program. Further research is required to measure the feasibility of this approach.

9.1.2 Soundness

C# programs run on Microsoft's .NET Framework. There are multiple implementations of the .NET Framework available, for example Dafny recommends Mono for Unix operating systems. Mono is an open source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime [4]. Mono offers support for various architectures, including x86, x86-64 (AMD64 and EM64T (64 bit)), ARM (little and big endian), and PowerPC.

Unfortunately, the Owicki-Gries method for verifying concurrent programs is unsound for weak memory models, even in the absence of auxiliary variables, as is shown in [12]. Even if we overcome the potential performance issues mentioned in Section 9.1.1, we may run into soundness issues due to the memory model of the underlying hardware. To garantuee soundness in a weak memory model, [12] proposes to strengthen the interference-freedom check. This could be implemented in our Dafny extension, possibly hidden behind a command-line flag. If a user does not specify this flag, we could show a warning during compilation to alert the user to potential soundness issues.

9.2 Optimization

9.2.1 Unnecessary Interference-Freedom Checks

Interference-freedom is trivially satisfied when two program components do not share variables. Thus, if the intersection of the set of variables of any two program components is empty, we know that these program components will not interfer with the assertions of the other program component. A possible optimization would be to detect this scenario and automatically skip the related intererence-freedom checks of the assertions and statements involved. In well-designed systems each program component will most likely only share variables with a small subset of other program components, so this could reduce the amount of interference-checks considerable. More research is required to test if this theoretical speed-up will be cost-effective in practise, because it could be that the detection is more expensive than the unnecessary but trivial checks given to the theorem prover (e.g. Z3).

[3] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/threading/thread-synchronization

[4] http://www.mono-project.com/

9.2.2 Heuristics

Some parallel program specifications have predictable standard proof outlines. For example, consider the one discussed in Section 2. To find the given standard proof outline, one approach is to start with the assertions for the program components that make them locally correct. This leads to:

```
        { x == 0 }
{ x == 0 }    { x == 0 }
x := x + 1    x := x + 2
{ x == 1 }    { x == 2 }
        { x == 3 }
```

To ensure all assertions interference-free, we then add the result of first executing the other parallel program component as a disjunction to each assertion:

```
                    { x == 0 }
{ x == 0 \/ x == 2 }    { x == 0 \/ x == 1 }
      x := x + 1                x := x + 2
{ x == 1 \/ x == 3 }    { x == 2 \/ x == 3 }
                    { x == 3 }
```

Heuristics could automatically add these assertions during verification to see if they are sufficient to proof initialization, finalization, local correctness and global correctness. This can be especially helpful if a parallel program has many program components or statements, since in examples such as the one above each additional program component or statement results in an additional disjunction to each assertion. For educational purposes such help is not necessary. In fact, it could be detrimental to a student's understanding of Owicki-Gries: finding the assertions that satisfy both local and global correctness is an important part of understanding the theory.

Appendices

Appendix A

- École normale supérieure
  - Course: Concurrency (2010)
  - Slides: Proof methods for concurrent programs - Owicki-Gries, Rely-Guarantee
- Eindhoven University of Technology
  - Paper: Incremental Verification of Owicki/Gries Proof Outlines Using PVS (2005)
- IBM Research
  - Slides: Parallelizing A Symbolic Compositional Model Checking Algorithm (2010)
- The IMDEA Software Institute
  - Paper: On the Strength of Owicki-Gries for Resources (2011)
- Imperial College London
  - Master Thesis: Reasoning About Concurrent Programs
- Katholieke Universiteit Leuven
  - Slides: Expressive Modular Fine-Grained Concurrency Specification (2011)
- Max Planck Institute for Software Systems
  - Paper: Owicki-Gries Reasoning for Weak Memory Models (2015)
  - Slides: Owicki-Gries for Weak Memory Models
- New York University | Courant Institute of Mathematical Sciences
  - Conference Slides: Construction of Owicki-Gries Proofs by Abstract Interpretation (2012)
- NICTA (National ICT Australia)
  - Paper: Controlled Owicki-Gries Concurrency: Reasoning about the Preemptible eChronos Embedded Operating System (2015)
- Nokia Bell Labs
  - Paper: Local Proofs for Global Safety Properties (2007)
- Technische Universität München
  - Paper: Owicki/Gries in Isabelle/HOL (1999)
  - Paper: Verifying Single and Multi-mutator Garbage Collectors with Owicki-Gries in Isabelle/HOL (2001)
  - Paper: Completeness of the Owicki-Gries System for Parameterized Parallel Programs (2001)
  - Dissertation: Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL (2001)

  - Presentation: Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL (2002)
  - Paper: Threader: A Constraint-Based Verifier for Multi-threaded Programs (2011)
- University of Cambridge
  - Slides: Independence and Concurrent Separation Logic (2006)
- University of Edinburgh
  - Paper: A generalization of Owicki-Gries's Hoare logic for a concurrent while language (1988)
  - Paper: Hoare Logic and Auxiliary Variables (1998)
  - Paper: Patterns for collaborative work in health care teams (2011)
- University of Ghana
  - Course: CSCD 417: Theory and Survey of Programming Languages
- University of Maryland
  - Course: Foundations of Software Verification (2015)
  - Slides: Owicki-Gries
- University of Passau
  - Paper: Owicki-Gries Method of Axiomatic Verification of Parallel, Shared-Memory Programs (2010)
- University of Pennsylvania
  - Course: CIS 673: Computer-Aided Verification, Fall 2016
  - Slides: Concurrency. Lectures by Arjun Radhakrishna, Lecture 2 (2016)
- The University of Queensland
  - Paper: Multiprogram Design in the theory of Owicki and Gries (2001)
  - Paper: Trace Semantics for the Owicki-Gries Theory Integrated with the Progress Logic from UNITY (2007)
- University of Toronto
  - Slides: Inductive Data Flow Graphs (2013)
- UNSW Sydney (The University of New South Wales)
  - Course: Software-Engineering Workshop Course (2014)
  - Slides: Summary of Owicki-Gries method

Bibliography

[1] Krzysztof Apt, Frank S De Boer, and Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs*. Springer Science & Business Media, 2010.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[3] Josh Berdine, Cristiano Calcagno, and Peter W O'hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[4] Stefan Blom and Marieke Huisman. The vercors tool for verification of concurrent programs. *FM*, 8442:127–131, 2014.

[5] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[6] Brijesh Dongol and Doug Goldson. Extending the theory of owicki and gries with a logic of progress. *arXiv preprint cs/0512012*, 2005.

[7] Cormac Flanagan, Stephen N Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(4):20, 2008.

[8] Doug Goldson and Brijesh Dongol. Concurrent program design in the extended theory of owicki and gries. In *Proceedings of the 2005 Australasian symposium on Theory of computing-Volume 41*, pages 41–50. Australian Computer Society, Inc., 2005.

[9] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *CAV*, volume 11, pages 412–417. Springer, 2011.

[10] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. In *International Conference on Computer Aided Verification*, pages 449–465. Springer, 2015.

[11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. *NASA Formal Methods*, 6617:41–55, 2011.

[12] Ori Lahav and Viktor Vafeiadis. Owicki-gries reasoning for weak memory models. In *International Colloquium on Automata, Languages, and Programming*, pages 311–323. Springer, 2015.

[13] Claire Le Goues, K Rustan M Leino, and Michał Moskal. The boogie verification debugger (tool paper). In *International Conference on Software Engineering and Formal Methods*, pages 407–414. Springer, 2011.

[14] K Rustan M Leino. This is boogie 2. *Manuscript KRML*, 178:131, 2008.

[15] K Rustan M Leino. Specification and verification of object-oriented software. *Engineering Methods and Tools for Software Safety and Security*, 22:231–266, 2009.

[16] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[17] K Rustan M Leino and Peter Müller. A basis for verifying multi-threaded programs. In *European Symposium on Programming*, pages 378–393. Springer, 2009.

[18] K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer, 2009.

[19] K Rustan M Leino and Valentin Wüstholz. The dafny integrated development environment. *arXiv preprint arXiv:1404.6602*, 2014.

[20] Christian Lengauer. Owicki-gries method of axiomatic verification. In *Encyclopedia of Parallel Computing*, pages 1401–1406. Springer, 2011.

[21] Richard J Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[22] Julio Mariño, Raúl NN Alborodo, Lars-Åke Fredlund, and Ángel Herranz. Synthesis of verifiable concurrent java components from formal models. *Software & Systems Modeling*, pages 1–35, 2017.

[23] Arjan J Mooij and Wieger Wesselink. Incremental verification of owicki/gries proof outlines using pvs. In *International Conference on Formal Engineering Methods*, pages 390–404. Springer, 2005.

[24] Carroll Morgan. Summary of owicki-gries method. SE2011 in 2014s1, may 2014.

[25] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016.

[26] Leonor Prensa Nieto. *Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL*. PhD thesis, PhD thesis, Technische Universität München, 2002.

[27] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4):319–340, 1976.