Verifying Software with Whiley

David J. Pearce

December 9, 2014

Contents

Chapter 1

Introduction

When we write a program, we have in mind some idea of what it is supposed to do. When we have finished our program, we might run it to see whether it appears to do the right thing. However, as anyone who has ever written a program will know: *this is not always enough!* Even if our program appears to work after a few tests, there is still a good chance it will go wrong for other inputs we have not yet tried. The question, then, is: *how can we be certain that our program is correct?*

In trying to determine whether our program is correct, our first goal is to determine precisely what it should do. In writing our program, we may not have had a clear idea of this from the outset. Therefore, we need to determine a *specification* for our program. This is a precise description of what the program should and should not do. Only with this can we begin to consider whether or not our program actually does the right thing.

1.1 Infamous Software Failures

Introduce some classic historical bugs, and emphasis that we want to reduce these as much as possible. There are lots of good examples, some of which are not coding failures but failures of understanding the requirements, etc.

Numerous important software systems have failed due to program bugs. Historic examples include the Therac-25 disaster where a computer-operated X-ray machine gave lethal doses to patients, the 1988 worm which wreaked havoc on the internet by exploiting a buffer overrun, and the (unmanned) Ariane 5 rocket which exploded shortly after launch because of an integer overflow (see this video and this list for more).

1.2 Programming Languages

Modern programming languages, such as Java and C#, eliminate fairly simple classes of error (so-called type errors) through the process of type checking; however, they cannot detect more complex problems, such as the potential for a divide-by-zero error. Type checking has been used for a long time in programming languages, historical examples of which include: ALGOL, Pascal, Modula-2, C, C++, Java, C# and more.

1.3 Whiley

The Whiley programming language has been in active development since 2009. The language was designed specifically to help the programmer eliminate bugs from his/her software. The key feature is that Whiley allows programmers to write *specifications* for their functions, which are then checked by the compiler. For example, here is the specification for the `max()` function which returns the maximum of two integers:

```
function max(int x, int y) => (int z)
// must return either x or y
ensures x == z || y == z
// return must be as large as x and y
ensures x <= z && y <= z:
    // implementation
    if x > y:
        return x
    else:
        return y
```

Here, we see our first piece of Whiley code. This declares a function called `max` which accepts two integers `x` and `y`, and returns an integer `z`. The body of the function simply compares the two parameters and returns the largest. The two **`requires`** clauses form the function's *post-condition*, which is a guarantee made to any caller of this function. In this case, the `max` function guarantees to return one of the two parameters, and that the return will be as large as both of them. In plain English, this means it will return the maximum of the two parameter values.

When verification is enabled the Whiley compiler will check that every function meets its specification. For our `max()` function, this means it will check that body of the function guarantees to return a value which meets the function's post-condition. To do this, it will explore the two execution paths of the function and check each one separately. If it finds a path which does not meet the post-condition, the compiler will report an error. In this case, the `max()` function above is implemented correctly and so it will find no errors. The advantage of providing specifications is that they can help uncover bugs and other, more serious, problems earlier in the development cycle. This leads to software which is both more reliable and more easily maintained (since the specifications provide important documentation).

1.3.1 Objectives

Part I

Verifying Software

Chapter 2

Specifying Functions and Data

In this chapter, we embark upon the first step in the process of establishing that a program is (in some sense) correct. That is, to determine exactly what our program is supposed to do. To this end, we will learn how to write *specifications* for our programs which precisely describe their behaviour.

In doing this, we must acknowledge that precisely describing a program's behaviour is extremely challenging and, oftentimes, we want only to specify some *important aspect* of its permitted behaviour. This can give us many of the benefits from specification without all of the costs.

While we will consider how to write specifications in this chapter, we will largely ignore the issue of determining whether our programs meet their specifications. Although this is an important part of the process, it is also challenging and we will require several chapters to fully explore how this is done.

Clarify that there are different levels of specification, ranging from the minimum to illustrate lack of runtime error to the complete specification that a function is correct.

2.1 Overview

Specifying a program in Whiley consists of at least two separate activities; firstly, we provide appropriate specifications (called *invariants*) for any data types we have defined; secondly, we provide specifications in the form of *pre-* and *post-conditions* for any functions or methods defined.

Of course, having specified our program, we want to establish that it meets its specification. Or, in other words, that every function and method meets its specification. This is itself quite a challenge and, furthermore, will often uncover critical flaws in our program. For now, we will ignore the question of whether our program meets its specification, and focus on the question of how you actually write a specification.

2.1.1 Preconditions and Postconditions

To specify a function or method in Whiley we must provide an appropriate *precondition* and *postcondition*. A precondition is a condition over the parameters of a function that is required to be true when the function is called. The body of the function can then use this to make assumptions about the possible values of the parameters. Likewise, a postcondition is a condition over the return values of a function which is required to be true after the function is called. As a very simple example, consider the following function which accepts a positive integer and returns a non-negative integer (i.e. a natural number):

```
function decrement(int x) => (int y)
// Parameter x must be greater than zero
requires x > 0
// Return must be greater or equal to zero
ensures y >= 0:
```

```
    //
    return x - 1
```

Here, the **requires** and **ensures** clauses define the function's precondition and postcondition. Informally, we can see that this function does meet its specification. Unfortunately, for larger programs it is very difficult to be sure that informal reasoning like this is correct. In later chapters, we will see how the Whiley system can automatically check that a function or method meets its specifications for us.

The Whiley compiler reasons about functions by exploring the different control-flow paths through their bodies. Furthermore, as it learns more about the variables used in the function, it automatically takes this into account. For example:

```
function abs(int x) => (int y)
// Return value cannot be negative
ensures y >= 0:
    //
    if x >= 0:
        return x
    else:
        return -x
```

The Whiley compiler verifies that the implementation of this function meets its specification. At this point, it is worth considering in more detail what this really means. Since the Whiley compiler performs verification at *compile-time*, it does not consider specific values when reasoning about a function's implementation. Instead, it considers all possible input values for the function which satisfy its precondition. In other words, when the Whiley compiler verifies a function's implementation meets its specification, this means it does so *for all possible input values*.

2.1.2 Data Type Invariants

The above illustrates a function specification given through explicit pre- and post-conditions. However, we may also employ *constrained types* to simplify it as follows:

```
type nat is (int n) where n >= 0
type pos is (int p) where p > 0

function f(pos x) => (nat n)
// Return must differ from parameter
ensures n != x:
    //
    return x-1
```

Here, the **type** declaration includes a **where** clause constraining the permissible values for the type ($ represents the variable whose type this will be). Thus, `nat` defines the type of non-negative integers (i.e. the natural numbers). Likewise, `pos` gives the type of positive integers and is implicitly a subtype of `nat` (since the constraint on `pos` implies that of `nat`). We consider that good use of constrained types is critical to ensuring that function specifications remain as readable as possible.

The notion of type in Whiley is more fluid than found in typical languages. In particular, if two types $\mathtt{T}_1$ and $\mathtt{T}_2$ have the same *underlying* type, then $\mathtt{T}_1$ is a subtype of $\mathtt{T}_2$ iff the constraint on $\mathtt{T}_1$ implies that of $\mathtt{T}_2$. Consider the following:

```
type anat is (int x) where x >= 0
type bnat is (int x) where 2*x >= x

function f(anat x) => bnat:
    return x
```

In this case, we have two alternate (and completely equivalent) definitions for a natural number (we can see that `bnat` is equivalent to `anat` by subtracting `x` from both sides). The Whiley compiler is able to reason that these types are equivalent and statically verifies that this function is correct.

2.2 Language of Specifications

2.2.1 Basics

2.2.2 Quantifiers

2.2.3 Functions

2.3 Weak vs Strong Specifications

2.4 Static vs Dynamic Checking

2.5 Partial vs Total Correctness

2.6 Exercises

Exercise 1 *The function `neg()` returns the arithmetic negation of a value. For example, `neg(1) = -1`. An implementation of this function is given as follows:*

```
function neg(int x) => (int r):
    return -x
```

Provide an appropriate post-condition *for this function.*

Exercise 2 *The `swap` function accepts to integers and returns them with their order swapped. The signature for the function is:*

```
function swap(int x, int y) => (int a, int b):
    ...
```

Provide an appropriate specification and implementation for this function.

Exercise 3 *A* natural *number is an integer which is greater-than-or-equal to zero. The following function adds three natural numbers together to produce a natural number:*

```
function sum3(int x, int y, int z) => (int r)
// No parameter can be negative
requires ...
// Return value cannot be negative
ensures ...:
    //
    return x + y + z
```

Complete the given **`requires`** *and* **`ensures`** *clauses based on the given English descriptions.*

Exercise 4 *The following function computes the absolute difference between two values:*

```
function diff(int x, int y) => (int r):
    //
    if x > y:
```

```
        return x - y
    else:
        return y - x
```

A pre-condition *of this function is that parameter* `x` *is between* `0` *and* `255` *(inclusive) and, likewise, that variable* `y` *is between* `-128` *and* `127` *(inclusive). Provide a partial specification for this function which constraints the ranges of the input and output variables as tightly as possible.*

Exercise 5 *The Gregorian calendar is the most widely used organisation of dates. A well-known saying for remembering the number of days in each month is the following:*

> *"Thirty days hath September, April, June and November. All the rest have thirty-one, except February which has twenty-nine ..."*

Note, in this exercise, we will ignore the issue of leap years. A simple function for returning a date can be defined as follws:

```
constant Jan is 1
constant Feb is 2
constant Mar is 3
constant Apr is 4
constant May is 5
constant Jun is 6
constant Jul is 7
constant Aug is 8
constant Sep is 9
constant Oct is 10
constant Nov is 11
constant Dec is 12

function getDate() => (int day, int month, int year):
    ...
```

Provide a specification for this function to ensure the returned date is valid (ignoring leap years). Furthermore, provide a simple implementation which meets this specification.

Exercise 6 *A well-known puzzle is that of the three water jugs. In this exercise, we will consider a cut down version of this which consists of two water jugs: a small jug (containing three litres) and a large jug (containing five litres). The goal is to complete the specification of the following function for pouring water from the small jug into the large jug:*

```
function pourSmall2Large(int smallJug, int largeJug) =>
                        (int smallJugAfter, int largeJugAfter)
// The small jug holds between 0 and 3 litres (before)
requires ...
// The large jug holds between 0 and 5 litres (before)
requires ...
// The small jug holds between 0 and 3 litres (after)
ensures ...
// The large jug holds between 0 and 5 litres (after)
ensures ...
// The amount in both jugs is unchanged by this function
ensures ...
// Afterwards, either the small jug is empty or the large jug is full
ensures ...:
    //
    if smallJug + largeJug <= 5:
        // indicates we're emptying the small jug
```

```
        largeJug = largeJug + smallJug
        smallJug = 0
    else:
        // indicates we're filling up the medium jug
        smallJug = largeJug + smallJug
        largeJug = 5
    // Done
    return smallJug, largeJug
```

Complete the missing **`requires`** *and* **`ensures`** *clauses based on the given English descriptions. Does the implementation meet the given specification?*

Chapter 3

Reasoning About Programs

In the previous Chapter, we learned how to write specifications for our programs in the form of pre-/post-conditions and invariants. The purpose of this was to allow us to state more clearly what our programs are supposed to do. An important step here lies in checking that our functions do indeed meet their specifications. In general, this is a tricky and time consuming process. Fortunately, the Whiley system takes much of the labour out of this process and can automatically that our programs meet their specifications.

In this chapter, we begin the process of understanding how to check that our programs meet their specifications. To this end, we consider two approaches: *forwards reasoning* and *backwards reasoning*. Whilst neither of these is strictly better than the other, you will most likely find forward reasoning to be more natural. Whilst this might seem to make backward reasoning redundant, we will find that it can still be very useful in Chapter 4. In this chapter, we will largely ignore functions which contains *loops* and/or are *recursive* because these constructs present additional challenges and will be discussed later in Chapters 4 and 5. This does not mean the programs considered in this chapter are uninteresting and, indeed, they may still contain a range of important constructs, including: *if statements*, *switch statements*, *list assignments*, *indirect assignments*, etc.

Also, introduce basic concept of a weakest precondition versus a strongest postcondition. Introduce control-flow graph and the path sensitive traversal?

3.1 Functions

As a very simple example, consider the following function which accepts a positive integer and returns a non-negative integer (i.e. a natural number):

```
function decrement(int x) => (int y)
// Parameter x must be greater than zero
requires x > 0
// Return must be greater or equal to zero
ensures y >= 0:
    //
    return x - 1
```

Here, the **requires** and **ensures** clauses define the function's precondition and postcondition. With verification enabled, the Whiley compiler will verify that the implementation of this function meets its specification. In fact, we can see this for ourselves by manually constructing an appropriate *verification condition* (that is, a logical condition whose truth establishes that the implementation meets its specification). In this case, the appropriate verification condition is `x > 0 ==> x-1 >= 0`. Unfortunately, although constructing a verification condition by hand was possible in this case, in general it's difficult if not impossible for more complex functions.

3.2 Assignments

Examine assignments.

3.3 Conditionals

Examine simple examples involving conditions, such as `abs()` and `max()`

3.4 Function Calls

Examine involving function calls

3.5 Compound Data Types

Examine examples with updating lists, records, etc.

3.6 Complex Reasoning

Examine problem with our simplistic way of thinking about assignments.

3.7 Forwards versus Backward Reasoning

Discuss the so-called weakest precondition transformer.

Chapter 4

Reasoning about Loops

4.1 Loop Invariants

4.2 Loop Variants

4.3 While Loops

- Where does the invariant hold?

4.4 Break Statements

4.5 Do/While Loops

4.6 Strategies

1. Weakening post-condition (see Gries, p195)
2. Extracting pre-condition from loop body
3. Lemmas (see Dafny tutorial on method lemmas)

4.7 Ghost Variables

4.8 Exercises

Exercise 7 *This is an example*

Chapter 5

Reasoning about Recursion

Basically repeat loop invariant stuff, but in context of recursion.

5.1 Induction

5.2 Lemmas

Part II

Understanding the Verification Process

Chapter 6

Verification Conditions

6.1 Assertion Language

6.2 Control-Flow Graphs

6.3 Hoare Logic

Appendix

Bibliography