Verifying Software with Whiley

David J. Pearce, 2015

Contents

Chapter 1

Introduction

Today, software is found in our cars, phones, aeroplanes, trains, power stations, medical equipment, military weapons and just about every other aspect of ours lives. Failures in software can have a significant effect on our lives. On Thursday August 14, 2003, a significant power outage spread across parts of the northeast and midwest of the United States, affecting an estimated 55 million people. During this time emergency services struggled to cope with the demand, backup generators failed, telephone networks overloaded, city water systems lost pressure and a state of emergency was declared in New York. In 2015, Air New Zealand announced that it would reboot the computers every three months on its new fleet of Boeing 787 *Dreamliners*. This was in response to software bug identified by Boeing (in fact, an *arithmetic overflow*) which can cause the US$250 million plane to lose all power mid-flight.

The rise of the internet has also given rise to the *black hats*. That is hackers who break into computer systems for criminal purposes, such as stealing credit card numbers. Many attacks on computer systems are made possible because of hidden software bugs. One such example was the infamous infamous *Heartbleed* bug which was disclosed in 2014. This was another simple software bug (in fact, a *buffer overrun*) found in the widely used OpenSSL cryptography library. This was concern across the internet for several reasons: firstly, a large number of people were affected (at the time, around half a million secure web servers were believed to be vulnerable); secondly, OpenSSL was an *open source* project but the bug had gone unnoticed for over two years. Joseph Steinberg of Forbes wrote, *"Some might argue that [Heartbleed] is the worst vulnerability found (at least in terms of its potential impact) since commercial traffic began to flow on the Internet".*

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).

1.1 Software Verification

Given the woeful state of much of the software we rely on every day, the question is: *what can we as computer scientists do about it?* Of course, we want to ensure that software when written is correct. *But how?* To answer this, we need to go back and rethink what softare development is. When we write a program, we have in mind some idea of what it should 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 is: *how can we be sure 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. If we have used a modern programming language, such as Java, C# or C++, then we are already familiar with this idea. These languages require a limited specification be given for functions in the form of *types*. That is, when writing a function in these languages we must specify the permitted type of each parameter and the return. These types put requirements on our code and ensure that certain errors are impossible. For example, when calling a function we must ensure that the argument values we give have the correct type. If we fail to do this, then the compiler will complain with an error message of some kind and force us to immediately *fix the problem*.

Software verification is the process of checking that a program meets its specification. In this book we adopt a specific approach referred to as *automated software verification*. This is where a tool is used to automatically check a function meets its specification or not. This is very similar to the way that compilers for languages like Java check that types are used correctly. The tool we choose for this is called *Whiley*. This programming language allows us to write specifications for our functions, and provides a special compiler which will attempt to check them for us automatically. We choose this tool because we are familiar with it, but it is not the only tool we could have chosen. Other suitable tools such as Spark/ADA, Dafny, Spec#, ESC/Java provide similar functionality and could also be used with varying degrees of success.

1.2 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).

1.2.1 Objectives

Part I

Verifying Software

Chapter 2

Specifying Programs

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.

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. 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.

Having specified our program, we want to establish that it meets its specification. Or, in other words, that every function and method meets its own specification. This is itself quite a challenge and, furthermore, will often uncover critical flaws in our program. 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.

2.1 Specifications as Contracts

A specification is a contract between two parties: the *client* and *supplier*. The client represents the person(s) using the given program, whilst the supplier is the person(s) who implemented it. The specification ties the *inputs* of the program to its *outputs* in two ways:

- **Inputs.** The specification states what is *required* of the inputs for the program to behave correctly. The client is responsible for ensuring the correct inputs are always given. If an incorrect input is given, the contract is broken and the program may do something unexpected.

- **Outputs.** The specification states what outputs must be *ensured* by a correctly behaving program. The supplier is responsible for ensuring all outputs meet the specification, assuming that correct inputs were provided.

From this, we can see that both parties in the contract have obligations they must meet. This also allows us to think about *blame*. That is when something goes wrong *who is responsible?* If the inputs to our program we incorrect according to the specification, we can blame the client. On the other hand if the inputs were correct, but the outputs were not, then we can blame the supplier.

Example. As a simple example, let us consider a function for finding the maximum values from an array of integer values. The following provides an *informal* specification for this function:

```
// REQUIRES: At least one item in items array
//
// ENSURES: Item returned was largest item in items array
function max([int] items) → (int item)
```

We have specified our function above using comments to document: firstly, the requirements needed for the inputs (i.e. that the array must have at least one element); and, secondly, the expectations placed on the outputs (i.e. that it returns the largest element in the array). Thus, we could not expect the call `max([])` to operate correctly; likewise, if the call `max([1,2])` returned `3` we would rightly say the implementation was incorrect. ■

Could talk about limitations of english language comments

2.2 Specifying Functions

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.

Example. As a very simple example, consider the following specification for our function which finds the maximum value from an array integers:

```
function max([int] items) => (int item)
// Must have one or more items
requires |items| > 0
// Value returned not smaller than anything in items
ensures all { i in 0 .. |items| | items[i] <= item }
// Value returns was in items array
ensures exists { i in 0 .. |items| | items[i] == item }
```

Here, the `requires` clause declares the function's precondition, whilst the `ensures` clauses declare its postcondition. This specification is largely the same as that given informally using comments before. However, we regard this specification as being *formal* because, for any set of inputs and outputs, we can calculate precisely whether the inputs or outputs were malformed. For example, consider the call `max([])`. We can say that the inputs to this call are incorrect, because `|[]| < 0` evaluates to `false`. For the informal version given above, we cannot easily evaluate the English comments to determine whether they were met or not. Instead, we rely on our human judgement for this — but, unfortunately, this can easily be wrong! ■

When specifying a function in Whiley, the `requires` clause(s) may only refer to the parameters, whilst the `ensures` clause(s) may also refer to the returns. Note, however, that the `ensures` clause(s) always consider those values of the parameters that held on entry to the function, not those which might hold at the end.

2.2.1 Meeting the Contract (Supplier)

We will now consider more precisely what it means for the supplier to meet the contract set out in a function's specification.

Example. To illustrate a function which does not meet its specification, consider the following incorrect implementation of our `max()` function:

```
function max([int] items) => (int item)
// Must have one or more items
requires |items| > 0
// Value returned not smaller than anything in items
ensures all {i in 0 .. |items| | items[i] <= item}
// Value returns was in items array
ensures exists {i in 0 .. |items| | items[i] == item}:
    //
    return items[0]
```

Hopefully, we can see this function does not do what we expect. But, let's think about this more formally. If the implementation is to be considered incorrect, *there must be at least one valid input which yields an incorrect output*. To show our implementation is incorrect, we must find such an input. Let us consider the input `[5]`, for which our implementation returns `5`. In this case, both `all {i in 0..|[5]| | [5][i] <= 5}` and `exists {i in 0..|[5]| | [5][i] == 5}` evaluate to `true` and, hence, *this input does not illustrate the problem*. In contrast, for the input `[1,4]` our implementation returns `1`. In this case, `all{ i in 0..|[1,4]| | [1,4][i] <= 1}` evaluates to `false` and, hence, *this input provides evidence that our implementation does not meet its contract*. ■

We say that a function *meets its specification* if it returns a valid output for all possible valid inputs. The challenge lies in knowing whether or not this is actually true. For

example, we could test some valid inputs to see whether they produce valid outputs. However, as Dijkstra said[?]:

> "*Program testing can be used to show the presence of bugs, but never to show their absence*"

What Dijkstra means here is that a single valid input is sufficient to show that a program does not meet its specification (as in our example above). However, to show that a function always meets its specification through testing requires testing *all possible inputs*. For some functions, this is actually possible in a feasible amount of time. Unfortunately, for most, it is not. For example, in a programming language like Java, a simple function accepting **`int`** parameter has 2^32 possible input values.

Example. We can now illustrate a correct implementation of our `max()` function as follows:

```
function max([int] items) => (int item)
// Must have one or more items
requires |items| > 0
// Value returned not smaller than anything in items
ensures all {i in 0 .. |items| | items[i] <= item}
// Value returns was in items array
ensures exists {i in 0 .. |items| | items[i] == item}:
   //
   int i = 0
   int r = items[0]
   //
   while i < |items:
      if r < items[i]:
          r = items[i]
      i = i + 1
   //
   return r
```

Understanding that this function does indeed meet is specification is not easy. Perhaps, intuitively, we can convince ourselves that it is using a number of arguments. For example, the variable `r` is only ever assigned values from the `items` array and, hence, the implementation must meet the second **`ensures`** clause of the specification. Similarly, since the loop iterates through and compares all elements in `items`, we believe the first **`ensures`** clause is met as well. But, again, *we are relying on our error-prone human judgement here*. ■

To be sure that a program meets its specification we must *prove* that it does. This means applying a mathematically rigorous mechanism which considers all possible input values.

Talk about challenge in proving a function is correct and how need to use Whiley for this. Briefly outline need for tools to do calculations for us, and how Whiley is one such example.

2.2.2 Meeting the Contract (Client)

- Multiple requires / ensures permitted
- Bodies of called functions ignored
- Pre/Post-conditions must be true before being called

2.3 Specifying Data

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 $\texttt{T}_1$ and $\texttt{T}_2$ have the same *underlying* type, then $\texttt{T}_1$ is a subtype of $\texttt{T}_2$ iff the constraint on $\texttt{T}_1$ implies that of $\texttt{T}_2$. 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.

2.4 Weak vs Strong Specifications

2.5 Static vs Dynamic Checking

2.6 Partial vs Total Correctness

2.7 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 post-condition. 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