Reflections on Verifying Software with Whiley

David J. Pearce and Lindsay Groves

School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
{djp,lindsay}@ecs.vuw.ac.nz

September 6, 2013

Abstract

An ongoing challenge for computer science is the development of a tool which automatically verifies programs meet their specifications, and are free from runtime errors such as divide-by-zero, array out-of-bounds and null dereferences. Several impressive systems have been developed to this end, such as ESC/Java and Spec#, which build on existing programming languages (e.g. Java, C#). Unfortunately, such languages were not designed for this purpose and this significantly hinders the development of practical verification tools for them. For example, soundness of verification in these tools is compromised. We have developed a programming language specifically designed for verification, called Whiley, and an accompanying verifying compiler. In this paper, we reflect on a number of challenges we have encountered in developing a practical system.

1 Introduction

The idea of verifying that a program meets a given specification for all possible inputs has been studied for a long time. Hoare's Verifying Compiler Grand Challenge was an attempt to spur new efforts in this area to develop practical tools [1]. A verifying compiler "*uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles*". Hoare's intention was that verifying compilers should fit into the existing development tool chain, "*to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components*". For example, commonly occurring errors could be automatically eliminated, such as: *division-by-zero*, *integer overflow*, *buffer overruns* and *null dereferences*.

The first systems that could be reasonably considered as verifying compilers were developed some time ago, and include that of King [2], Deutsch [3], the Gypsy Verification Environment [4] and the Stanford Pascal Verifier [5]. Following on from these, was the Extended Static Checker for Modula-3 [6]. Later, this became the Extended Static Checker for Java (ESC/Java) — a widely acclaimed and influential work in this area [7]. Building on this success was the Java Modeling Language (and its associated tooling) which provided a standard notation for specifying functions in Java [8]. More recently, the Spec# language [9, 10, 11] was developed on top of C#, whilst Dafny was developed from scratch to simplify verification [12, 13].

Continuing this line of work, we are developing a verifying compiler for the Whiley programming language [14, 15, 16, 17, 18]. Whiley is an imperative language designed to simplify verification. For example, Whiley uses unbound integer and rational arithmetic in place of e.g. IEEE 754 floating point (which is notoriously difficult to reason about [19]). Likewise, pure (i.e. mathematical) functions are distinguished from those which may have side-effects. Our goal is to develop a verifying compiler which can automatically establish a Whiley program as: *correct with respect to its declared specifications*; and, *free from runtime error* (e.g. divide-by-zero, array index-out-of-bounds, etc). More complex properties, such as establishing termination, are not considered (although would be interesting future work). Finally, the Whiley verifying compiler is released under an open source license (BSD), can be downloaded from `http://whiley.org` and forked at `http://github.com/DavePearce/Whiley/`.

Contribution. The seminal works by Floyd [20], Hoare [21], Dijkstra [22], and others provide a foundation upon which to develop tools for verifying software. However, in developing a verifying compiler for Whiley, we have encountered some gaps between theory and practice. In this paper, we reflect on our experiences using Whiley to verify programs and, in particular, highlight a number of challenges we encountered.

2 Language Overview

We begin by exploring the Whiley language and highlight some of the choices made in its design. For now, we stick to the basic issues of syntax, semantics and typing and, in the following section, we will focus more specifically on using Whiley for verification. Perhaps one of our most important goals was to make the system as accessible as possible. To that end, the language was designed to superficially resemble modern imperative languages (e.g. Python), and this decision has significantly affected our choices.

Overview. Languages like Java and C# permit arbitrary side-effects within methods and statements. This presents a challenge when such methods may be used within specifications. Systems like JML and Spec# require that methods used in specifications are *pure* (i.e. side-effect free). An important challenge here is the process of checking that a function is indeed pure.

A significant body of research exists on checking functional purity in object-oriented languages (e.g. [23, 24, 25, 26]). Much of this relies on interprocedural analysis, which is too costly for a verifying compiler. For the remainder, the ability to handle existing library code effectively is severely limited. In Java, for example, even simple methods such as `Object.equals()` cannot easily be shown as pure. This is because classes in the standard library have side-effecting `equals()` implementations [23]. To address this, Whiley is a hybrid object-oriented and functional language which divides into *a functional core* and an *imperative outer layer*. Everything in the functional core can be modularly checked as being side-effect free. To make this possible, Whiley incorporates first-class sets, lists and maps which are *values* (rather than mutable objects) and, hence, allow call-by-value semantics (more on this below).

Value Semantics. The prevalence of pointers — or references — in modern programming languages (e.g. Java, C++, C#) has been a major hindrance in the development of verifying compilers. Indeed, Mycroft recently argued that (unrestricted) pointers should be "considered harmful" in the same way that Dijkstra considered goto harmful [27]. To address this, all compound structures in Whiley (e.g. lists, sets, and records) have *value semantics*. This means they are passed and returned by-value (as in Pascal, MATLAB or most functional languages). But, unlike functional languages (and like Pascal), values of compound types can be updated in place. Whilst this latter point may seem unimportant, it serves a critical purpose: to give Whiley the appearance of a modern *imperative* language when, in fact, the functional core of Whiley is pure. This goes towards our goal of making the language as accessible as possible.

Value semantics implies that updates to a variable only affect that variable, and that information can only flow out of a function through its return value. Consider:

```
int f([int] xs):
    ys = xs
    xs[0] = 1
    ...
```

The semantics of Whiley dictate that, having assigned `xs` to `ys` as above, the subsequent update to `xs` does not affect `ys`. Arguments are also passed by value, hence `xs` is updated inside `f()` and this does not affect `f`'s caller. That is, `xs` is not a *reference* to a list of **`int`**; rather, it *is* a list of **`int`**s and assignments to it do not affect state visible outside of `f()`.

Whilst this approach may seem inefficient, a variety of techniques exist (e.g. reference counting) to ensure efficiency (see e.g. [28, 29, 30]). Indeed, the underlying implementation does pass compound structures by reference and copies them only when absolutely necessary.

Unbound Arithmetic. Modern languages typically provide fixed-width numeric types, such as 32bit twos-compliment integers, or 64-bit IEEE 754 floating point numbers. Such data types are notoriously difficult for an automated theorem prover to reason about [19]. Systems like JML and Spec# assume (unsoundly) that numeric types do not overflow or suffer from rounding. To address this, Whiley employs *unbounded integers* and *rationals* in place of their fixed-width alternatives and, hence, does not suffer the limitations of soundness discussed above.

Whilst performance is a concern here, Whiley is often able to encode an integer using, for example, a single 32bit twos-complement **`int`**. This is achieved by exploiting the specifications that are an integral part of Whiley programs. When the compiler can prove an integer will remain within certain bounds, it is free to use a fixed-width type. For example, every `string` in Whiley is a list of Unicode characters, whose range is between 0 and 1114111 — hence, a character can always be encoded using a single 32bit twos-complement integer.

Flow Typing. An unusual feature of Whiley is the use of a *flow typing system* (see e.g. [31, 32, 17, 18]). This gives Whiley the look-and-feel of a dynamically typed language (e.g. Python). Furthermore, automatic variable retyping through conditionals is supported using the **`is`** operator (similar to `instanceof` in Java) as follows:

```
define Circle as {int x, int y, int radius}
define Rect as {int x, int y, int width, int height}
define Shape as Circle | Rect

real area(Shape s):
    if s is Circle:
        return PI * s.radius * s.radius
    else:
        return s.width * s.height
```

A `Shape` is either a `Rect` or a `Circle` (which are both record types). The type test "`s` **`is`** `Circle`" determines whether `s` is a `Circle` or not. Unlike Java, Whiley automatically retypes `s` to have type `Circle` (resp. `Rect`) on the true (resp. false) branches of the **`if`** statement. There is no need to explicitly cast variable `s` to the appropriate `Shape` before accessing its fields.

Union Types. Another unusual feature of Whiley is the use of *union types* (see e.g. [33, 34]), which complement the flow type system. Consider the following example:

```
int|null indexOf(string str, char c):
   ...

[string] split(string str, char c):
   idx = indexOf(str,c)
   // idx has type null|int
   if idx is int:
       // idx now has type int
       below = str[0..idx]
       above = str[idx..]
       return [below,above]
   else:
       // idx now has type null
       return [str]
```

Here, `indexOf()` returns the first index of a character in the string, or **`null`** if there is none. The type **`int`**`|`**`null`** is a union type, meaning it is either an **`int`** *or* **`null`**. The system seamlessly ensures **`null`** is never dereferenced because the type **`null`**`|`**`int`** cannot be treated as an **`int`**. Instead, one must first check it *is* an **`int`** using e.g. "`idx` **`is int`**".

Null references have provided a significant source of error in languages like Java and C#, and their creation has been claimed as a billion dollar mistake [35]. The issue is that, in such languages, one can treat *nullable* references as though they are *non-null* references [36]. However, many solutions have been proposed (e.g. [37, 38, 39, 40, 41]), we find that Whiley's union types provide an elegant solution.

Negation Types. Retyping variables after type tests necessitates the use of *negation types* (see e.g. [42]). A negation type `!T` defines the set of values *not* in `T`. The following example (albeit artificial) illustrates:

```
int f(any x):
    if x is {any field}:
        return 0
    else if x is {int field}:
        return 1
    else:
        return 2
```

The above code does not type check. This is because the type of `x` on the false branch of the first condition is `!{any field}`. Furthermore, `{int field}` is a subtype of `{any field}` and, hence, *not* a subtype of `!{any field}`. Therefore, the compiler emits an error message stating that "`return 1` is unreachable".

Effective Unions. A union of types of the same kind (e.g. a union of record types, or a union of list types) can expose commonality and are called *effective unions* (e.g. an effective record type). In the case of a union of records, fields common to all records are exposed. The following (simplified) excerpt, taken from a PNG image decoder, illustrates:

```
define IHDR_TYPE as 0x52444849
define IHDR as {int type,int width,int height}
define IDAT as {int type,[byte] data}
define CHUNK as IHDR | IDAT

[CHUNK] decode([byte] data):
   chunks = []
   pos = 0
   // parse chunks
   while pos < |data|:
      chunk,pos = parse(data,pos)
      chunks = chunks + [chunk]
   // check first chunk
   assert |chunks| > 0 && chunks[0].type == IHDR_TYPE
   // done
   return chunks

CHUNK parse([byte] data, int pos):
    ...
```

A PNG file defines a list of `CHUNK`s, where the first `CHUNK` must be an instance of `IHDR`. The above simply decodes the bytes of a PNG file into the list of `CHUNK`s. It also checks the first `CHUNK` is an `IHDR`.

In the above, the expression `chunks[0]` has union type `IHDR|IDAT` (a.k.a type `CHUNK`). The record types `IHDR` and `IDAT` differ and are not related through subtyping. However, they share a common field, namely `type`. Thus, `IHDR|IDAT` defines an *effective record type* of `{int type, ...}`

which means `chunks[0].type` is considered type safe. Finally, it's interesting to note that the notion of an effective record type is similar, in some ways, to that of the *common initial sequence* found in C [43].

Recursive Data Types. Whiley provides recursive types which are similar to the abstract data types found in functional languages (e.g. Haskell, ML, etc). For example:

```
define LinkedList as null | {int data, LinkedList next}

int length(LinkedList l):
  if l is null:
    return 0 // l now has type null
  else:
    return 1 + length(l.next) // l now has type {int data, LinkedList next}
```

Here, we again see how flow typing gives an elegant solution. More specifically, on the false branch of the type test "`l is null`", variable `l` is automatically retyped to `{int data, LinkedList next}` — thus ensuring the subsequent dereference of `l.next` is safe. No casts are required as would be needed for a conventional imperative language (e.g. Java). Finally, like all compound structures, the semantics of Whiley dictates that recursive data types are passed by value (or, at least, appear to be from the programmer's perspective).

Executable Specifications. Verifying compilers employ automated theorem provers to check programs correctness. Unfortunately, undecidability and other practical constraints (e.g. timeouts) present a challenge here, and we must anticipate that the theorem prover may not always reach a conclusion. To deal with this, systems like JML and Spec# permit the use of *runtime checking* as an alternative mechanism [8, 44]. This is a useful fall-back when the theorem prover cannot make strong conclusions about the validity of a statement. Spec#, for example, inserts runtime checks where necessary, and subsequently removes them if the theorem prover demonstrates it is safe to do so.

To use runtime checking in this manner, one must ensure that all specifications are *executable* (i.e. can be turned into runtime checks). In particular, the ability to perform *unbound quantification* can easily break this requirement [45, 46]. The following example illustrates (using Whiley-like syntax):

```
int f(int x, int y):
    return a+b+c

assert all { x in int, y in int | f(x,y) == f(y,x) }
```

The **`assert`** statement requires that the function `f(int,int)` is *commutative*. This example cannot be expressed in Whiley, because it is quantifying over the (infinite) set of all integers. As such, the assertion *cannot be converted into a runtime check* — because it would require an infinite amount of time to check[1]. In general, any attempt to quantify over an infinite set is referred to as unbounded quantification. Unfortunately, both JML and Spec# permit unbound quantification which prohibits the use of runtime checks in some cases [44, 9].

Whiley provides a strong guarantee every specification can be turned into a runtime check. In particular, all quantification must be *bounded* as the following illustrates:

```
int sum([int] xs)
    requires all { x in xs | x >= 0}, ensures $ >= 0:
    ...
```

This gives a specification for the `sum()` function which accepts a list of natural numbers (i.e. non-negative integers) and produces a natural number. The quantification is bounded by the parameter `xs` and, hence, can be easily translated into a **`for`**-loop over its elements.

[1]Whilst it is true that integers in most languages have fixed-width representations, this means only that quantification over integers is implicitly bounded in such languages. However, for other data types (e.g. collections), this is not the case.

The choice to restrict Whiley to executable specifications means the language is strictly less expressive than, for example, JML. However, it does ensure a smooth transition to runtime checking when the theorem prover cannot be conclusive about a given verification condition. More specifically: for each verification condition, either the automated theorem prover shows there *definitely is an error*, or it shows there *definitely is not an error*, or it cannot draw a strong conclusion and, hence, *there maybe an error*. Only in the latter case does the compiler resort to using a runtime check — and it emits a warning to indicate this.

3 Verification

A key goal of the Whiley project is to develop an open framework for research in automated software verification. As such, we now explore verification in Whiley.

Example 1 — Preconditions and Postconditions. The following Whiley code defines a function accepting a positive integer and returning a non-negative integer (i.e. natural number):

```
int f(int x) requires x > 0, ensures $ >= 0 && $ != x:
    return x-1
```

Here, the function `f()` includes a **`requires`** and **`ensures`** clause which correspond (respectively) to its *precondition* and *postcondition*. In this context, \$ represents the return value, and must be used in the **`ensures`** clause. The Whiley compiler statically verifies that this function meets its specification.

A slightly more unusual example is the following:

```
int f(int x) requires x >= 0, ensures 2*$ >= $:
    return x
```

In this case, we have two alternate (and completely equivalent) definitions for a natural number. We can see that the precondition is equivalent to the postcondition by subtracting \$ from both sides. The Whiley compiler is able to reason that these are equivalent and statically verifies that this function is correct.

Example 2 — Conditionals. Variables in Whiley are described by their underlying type and those constraints which are shown to hold. As the automated theorem prover learns more about a variable, it automatically takes this into consideration when checking constraints are satisfied. For example:

```
int abs(int x) ensures $ >= 0:
    if x >= 0:
        return x
    else:
        return -x
```

The Whiley compiler statically verifies that this function always returns a non-negative integer. This relies on the compiler to reason correctly about the implicit constraints implied by the conditional. A similar, but slightly more complex example is that for computing the maximum of two integers:

```
int max(int x, int y)
        ensures $ >= x && $ >= y && ($==x || $==y):
    if x > y:
        return x
    else:
        return y
```

Again, the Whiley compiler statically verifies this function meets its specification. Here, the body of the function is almost completely determined by the specification — however, in general, this it not the case.

Example 3 — Bounds Checking. An interesting example which tests the automated theorem prover more thoroughly is the following:

```
null|int indexOf(string str, char c):
    for i in 0..|str|:
        if str[i] == c:
            return i
    return null
```

In this case, the access `str[i]` must be shown as within the bounds of the list `str`. Here, the range constructor `X..Y` returns a list of consecutive integers from `X` upto, but not including `Y` (and, futhermore, if `X >= Y` then the empty list is returned). Hence, this function cannot cause an out-of-bounds error and the Whiley compiler statically verifies this.

In fact, the specification for `indexOf()` could be made more precise as follows:

```
null|int indexOf(string str, char c)
    ensures $ == null || (0 <= $ && $ < |str|):
    ...
```

In this case, we are additionally requiring that, when the return value is an **`int`**, then it is a valid index into `str`. Again, the Whiley compiler statically verifies this is the case.

Example 4 — Loop Invariants. Another example illustrates the use of *loop invariants* in Whiley:

```
int sum([int] list)
        requires all { item in list | item >= 0 },
        ensures $ >= 0:
    r = 0
    for v in list where r >= 0:
        r = r + v
    return r
```

Here, a bounded quantifier is used to enforce that `sum()` accepts a list of natural numbers. A key constraint is that summing a list of natural numbers yields a natural number (recall arithmetic is unbounded and does not overflow in Whiley). The Whiley compiler statically verifies that `sum()` does indeed meet this specification. The loop invariant is necessary to help the compiler generate a sufficiently powerful verification condition to prove the function meets the post condition (more on this later).

Example 5 — Recursive Structures. The Whiley language supports invariants over recursive structures, as the following illustrates:

```
define Tree as null | Node

define Node as { int data, Tree lhs, Tree rhs } where
            (lhs == null || lhs.data < data) &&
            (rhs == null || rhs.data > data)
```

This defines something approximating the notion of an unbalanced binary search tree. Unfortunately, the invariant permits e.g. `data < lhs.rhs.data` for a given tree node and, thus, is not sufficient to properly characterise binary search trees. Whilst our focus so far has been primarily on array programs and loop invariants, in the future we plan to place more emphasis on handling recursive structures, such as binary search trees.

4 Hoare Logic

We now briefly review Hoare logic [21] and the weakest precondition transformer, before examining in §5 a number of challenges we encountered in practice. Hoare logic provides some important

$$\frac{}{\left\{p[x/e]\right\}\ \mathtt{x} = \mathtt{e}\ \left\{p\right\}}\ (\text{H-Assign}) \qquad \frac{\left\{p\right\}\ \mathtt{s}_1\ \left\{r\right\}\quad \left\{r\right\}\ \mathtt{s}_2\ \left\{q\right\}}{\left\{p\right\}\ \mathtt{s}_1;\mathtt{s}_2\ \left\{q\right\}}\ (\text{H-Sequence})$$

$$\frac{\begin{array}{c}\left\{p_1\right\}\ \mathtt{S}\ \left\{q_1\right\}\\ \mathtt{p}_2 \implies \mathtt{p}_1\quad \mathtt{q}_1 \implies \mathtt{q}_2\end{array}}{\left\{p_2\right\}\ \mathtt{s}\ \left\{q_2\right\}}\ (\text{H-Consequence}) \qquad \frac{\begin{array}{c}\left\{p \land e_1\right\}\ \mathtt{s}_1\ \left\{q\right\}\\ \left\{p \land \neg e_1\right\}\ \mathtt{s}_2\ \left\{q\right\}\end{array}}{\left\{p\right\}\ \mathtt{if}\ \mathtt{e}_1\mathtt{:}\ \mathtt{s}_1\ \mathtt{else:}\ \mathtt{s}_2\ \left\{q\right\}}\ (\text{H-If})$$

$$\frac{\left\{e_1 \land e_2\right\}\ \mathtt{s}\ \left\{e_2\right\}}{\left\{e_2\right\}\ \mathtt{while}\ \mathtt{e}_1\ \mathtt{where}\ \mathtt{e}_2\mathtt{:}\ \mathtt{s}\ \left\{\neg e_1 \land e_2\right\}}\ (\text{H-While})$$

Figure 1: Hoare Logic.

background to understanding how the Whiley verifying compiler works, and why certain difficulties manifest themselves. Our discussion here is necessarily brief and we refer to Frade and Pinto for an excellent survey [47].

4.1 Overview

The rules of Hoare logic are presented as judgements involving triples of the form: $\left\{p\right\}\ \mathtt{s}\ \left\{q\right\}$. Here, p is the precondition, s the statement to be executed and q is the postcondition. Figure 1 presents the rules of Hoare Logic which, following Whiley, we have extended to include explicit loop invariants. To better understand these rules, consider the following example:

$$\left\{\mathtt{x} \geq \mathtt{0}\right\}\ \mathtt{x} = \mathtt{x} + \mathtt{1}\ \left\{\mathtt{x} > \mathtt{0}\right\}$$

Here we see that, if $\mathtt{x} \geq \mathtt{0}$ holds immediately before the assignment then, as expected, it follows that $\mathtt{x} > \mathtt{0}$ holds afterwards. However, whilst this is intuitively true, it is not so obvious how this triple satisfies the rules of Figure 1. For example, as presented it does not immediately satisfy H-Assign. However, rewriting the triple is helpful here:

$$\left\{\mathtt{x} + \mathtt{1} > \mathtt{0}\right\}\ \mathtt{x} = \mathtt{x} + \mathtt{1}\ \left\{\mathtt{x} > \mathtt{0}\right\}$$

The above triple clearly satisfies H-Assign and, furthermore, we can obtain the original triple from it via H-Consequence (i.e. since $x + 1 > 0 \implies x \geq 0$). The following illustrates a more complex example:

```
int f(int i) requires i >= 0, ensures $ >= 10:
  {i ≥ 0}
  while i < 10 where i >= 0:
      {i < 10 ∧ i ≥ 0}
      i = i + 1
      {i ≥ 0}
  {i ≥ 10 ∧ i ≥ 0}
  return i
```

Here, we have provided the intermediate assertions which tie the Hoare triples together. In essence, we have verified that this function meets its specification. Note that this does not prove the function terminates (although we can see that it does) and, in general, the Whiley system does not establish termination (although this is interesting future work).

4.2 Verification Condition Generation

Automatic program verification is normally done with a *verification condition generator* [7]. This converts the program source into a series of logical conditions — called *verification conditions* — to be checked by the automated theorem prover. There are two approaches: propagate *forward* from the precondition; or, propagate *backwards* from the postcondition. We now briefly examine these in more detail.

Weakest Preconditions. Perhaps the most common way to generated verification conditions is via the *weakest precondition transformer* [48]. This determines the weakest precondition (written $wp(\mathtt{s},\mathtt{q})$) that ensures a statement s meets a given postcondition q. Roughly speaking, this corresponds to propagating the postcondition backwards through the statement. For example, consider verifying this triple:

$$\Big\{\mathtt{x} \geq 0\Big\}\ \mathtt{x} = \mathtt{x} + 1\ \Big\{\mathtt{x} > 0\Big\}$$

Propagating $\mathtt{x} > 0$ backwards through $\mathtt{x} = \mathtt{x} + 1$ gives $\mathtt{x} + 1 > 0$ via H-Assign. From this, we can generate a verification condition to check the given precondition implies this calculated weakest precondition (i.e. $\mathtt{x} \geq 0 \implies \mathtt{x} + 1 > 0$). To understand this process better, let's consider a more concrete example:

```
int f(int x) requires x >= 0, ensures $ >= 0:
    x = x - 1
    return x
```

The implementation of this function does not satisfy its specification. Using weakest preconditions to determine this corresponds to the following chain of reasoning:

$$\begin{array}{l} \mathtt{x} \geq 0 \implies wp(\mathtt{x} = \mathtt{x} - 1, \mathtt{x} \geq 0) \\ \hookrightarrow \mathtt{x} \geq 0 \implies \mathtt{x} - 1 \geq 0 \\ \hookrightarrow \bot \end{array}$$

Here, the generated verification condition is $\mathtt{x} \geq 0 \implies wp(\mathtt{x} = \mathtt{x} - 1, \mathtt{x} \geq 0)$. This is then reduced to a contradiction (e.g. by the automated theorem prover) which indicates the original program did not meet its specification.

Strongest Postconditions. By exploiting Floyd's rule for assignment [20], an alternative formulation of Hoare logic can be developed which propagates in a forward direction and, thus, gives a *strongest postcondition transformer* [49, 47]. This determines the strong postcondition (written $sp(\mathtt{p},\mathtt{s})$) that holds after a given statement s with pre-condition p. For example, propagating $\mathtt{x} = 0$ forwards through $\mathtt{x} = \mathtt{x} + 1$ yields $\mathtt{x} = 1$. Using strongest postconditions to verify functions is similar to using weakest preconditions, except operating in the opposite direction. Thus, for a triple $\{\mathtt{p}\}\ \mathtt{s}\ \{\mathtt{q}\}$, we generate the verification condition $sp(\mathtt{p}, \mathtt{s}) \implies \mathtt{q}$. For example, consider:

$$\Big\{\mathtt{x} = 0\Big\}\ \mathtt{x} = \mathtt{x} + 1\ \Big\{\mathtt{x} > 0\Big\}$$

In this case, the generated verification condition will be $\mathtt{x} = 1 \implies \mathtt{x} > 0$ (which can be trivially established by an automated theorem prover).

5 Experiences

In the previous section, we outlined the process of automatic verification using Hoare logic. This was the starting point for developing our verifying compiler for Whiley. However, whilst Hoare logic provides an excellent foundation for reasoning about programs, there remain a number of hurdles to overcome in developing a practical tool. We now reflect on our experiences in this endeavour using examples based on those we have encountered in practice.

5.1 Loop Invariants

The general problem of automatically determining loop invariants is a hard algorithmic challenge (see e.g. [50, 51, 52]). However, we want to cover as many simple cases as possible to reduce programmer burden. We now examine a range of simple cases that, in our experience, appear to occur frequently.

Loop Invariant Variables. From the perspective of a practical verification tool, the rule H-WHILE from Figure 1 presents something of a hurdle. This is because it relies on the programmer to completely specify the loop invariant *even in cases where this is unnecessary*. For example, consider the following Whiley program:

```
int f(int x) requires x > 0, ensures $ >= 10:
    i = 0
    while i < 10 where i >= 0:
        i = i + x
    return i
```

Intuitively, we can see this program satisfies its specification. Unfortunately, this program cannot be shown as correct under the rules of Figure 1 because the loop invariant is too weak. Unfortunately, rule H-WHILE only considers those facts given in the loop condition and the declared loop invariant — hence, all information about x is discarded. Thus, under H-WHILE, the verifier must assume that x could be negative within the loop body — which may seem surprising because x is not modified by the loop!

To verify this program under rule H-WHILE, the loop invariant must be strengthened as follows:

```
int f(int x) requires x > 0, ensures $ >= 10:
    i = 0
    while i < 10 where i >= 0 && x >= 0:
        i = i + x
    return i
```

Now, one may say the programmer made a mistake here in not specifying the loop invariant well enough; however, our goal in developing a practical tool is to reduce programmer effort as much as possible. Therefore, in the Whiley verifying compiler, *loop invariant variables are handled automatically so that the programmer does not need to respecify their invariants*.

Simple Synthesis. As mentioned above, generating loop invariants in the general case is hard. However, there are situations where loop invariants can easily be determined. The following illustrates an interesting example:

```
int sum([int] xs)
      requires all { x in xs | x >= 0 }, ensures $ >= 0:
   i = 0
   r = 0
   while i < |xs|:
       r = r + xs[i]
       i = i + 1
   return r
```

This function computes the sum of a list of natural numbers, and returns a natural number. The question to consider is: *did the programmer specify the loop invariant properly?* Unfortunately, the answer again is: *no*. In fact, a loop invariant needs to be given explicitly as follows:

```
   ...
   while i < |xs| where i >= 0 && r >= 0:
       r = r + xs[i]
       i = i + 1
   return r
```

Here, we can see that an explicit loop invariant has been given through a `where` clause. The need for this is frustrating as, intuitively, it is trivial to see that it holds. In the future, we hope to automatically synthesize simple loop invariants such as this.

Constrained Types. Whiley also supports the notion of a *constrained type* as follows:

```
define nat as int where $ >= 0
```

Here, the **`define`** statement 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)

An interesting aspect of Whiley's design is that local variables are not explicitly declared. This gives Whiley the look-and-feel of a dynamically typed language and goes towards our goal of making the language accessible. In fact, permitting variable declarations would provide an alternative solution to the above issue with `sum()`:

```
int sum([int] xs)
      requires all { x in xs | x >= 0 }, ensures $ >= 0:
   nat i = 0
   nat r = 0
   while i < |xs|:
       r = r + xs[i]
       i = i + 1
   return r
```

Here, variable declarations are used to restrict the permitted values of variables `i` and `r` throughout the function. Unfortunately, at the moment, the Whiley language does not permit local variable declarations and, hence, the above is invalid. In the future, we plan to support local variable declarations for this purpose.

Loop Invariant Properties. Whilst our verifying compiler easily handles loop invariant variables, there remain situations when invariants need to be needlessly respecified. Consider the following:

```
[int] add([int] v1, [int] v2)
      requires |v1| == |v2|, ensures |$| == |v1|:
    i = 0
    while i < |v1| where i >= 0:
        v1[i] = v1[i] + v2[i]
        i = i + 1
    return v1
```

This example adds two vectors of equal size. Unfortunately, this again does not verify under the rule H-WHILE because the loop invariant is too weak. The key problem is that `v1` is modified in the loop and, hence, our above solution for loop invariant variables does not apply. Following rule H-WHILE, the verifying compiler can only reason about what is specified in the loop condition and invariant. Hence, it knows nothing about the size of `v1` after the loop. This means, for example, it cannot establish that `|v1| == |v2|` holds after the loop. Likewise (and more importantly in this case), it cannot establish that the size of `v1` is unchanged by the loop (which we refer to as a *loop invariant property*). Thus, it cannot establish that the size of the returned vector equals that held in `v1` on entry, and reports the function does not meet its postcondition.

Unfortunately, in the Whiley language developed thus far, it is *impossible to specify that the size of `v1` is unchanged by the loop*. This is because it requires the notion of a *loop variant*. The following illustrates a hypothetical syntax for this in Whiley:

```
    ...
    while i < |v1| where i >= 0 && |v1'| == |v1|:
        v1[i] = v1[i] + v2[i]
        i = i + 1
    return v1
```

Here, `v1'` respents the value of `v1` on the previous iteration. Unfortunately, this syntax is not yet supported in Whiley (although it is planned for the future).

Finally, it is possible to specify a loop invariant which allows the above function to be verified by our compiler. Since `v2` is not modified by the loop, and `|v1| == |v2|` held on entry, we can use `i >= 0 && |v1| == |v2|` as the loop invariant.

Overriding Invariants. In most cases, the loop condition and invariant are used independently to increase knowledge. However, in some cases, they need to be used in concert. The following illustrates:

```
[int] create(int count, int value)
    requires count >= 0, ensures |$| == count:
   r = []
   i = 0
   while i < count:
      r = r + [value]
      i = i + 1
   return r
```

This example uses the list append operator (i.e. `r + [value]`) and is surprisingly challenging. An obvious approach is to connect the size of `r` with `i` as follows:

```
   ...
   while i < count where |r| == i:
      r = r + [value]
      i = i + 1
   return r
```

Unfortunately, this is insufficient under the rule H-WHILE from Figure 1. This is because, after the loop is complete, the rule establishes the invariant and the *negated* condition. Thus, after the loop, we have $\texttt{i} \geq \texttt{count} \wedge |\texttt{r}| == \texttt{i}$, but this is insufficient to establish that $|\texttt{r}| == \texttt{count}$. In fact, we can resolve this by using an *overriding loop invariant* as follows:

```
   ...
   while i < count where i <= count && |r| == i:
      r = r + [value]
      i = i + 1
   return r
```

In this case, $\texttt{i} \geq \texttt{count} \wedge \texttt{i} \leq \texttt{count} \wedge |\texttt{r}| == \texttt{i}$ holds after the loop, and the automated theorem prover will trivially establish that $|\texttt{r}| == \texttt{count}$.

5.2 Error Reporting

Error reporting is an important practical consideration for any verification tool, as we want error messages which are as meaningful, and precise, as possible. We now consider how the two approaches to verification condition generation affect this.

Weakest Preconditions. An unfortunate side-effect of operating in a backwards direction, as $wp(\texttt{s},\texttt{q})$ does, is that reporting useful errors in the source program is more difficult. For example, consider this simple example which performs an integer division:

```
int f(int x) requires x > 0, ensures $ > 0:
    x = 1 / (x - 1)
    return x
```

This function contains a bug which can cause a division-by-zero failure (i.e. if `x==1` on entry). Using $wp(\texttt{s},\texttt{q})$, a single verification condition is generated for this example:

$$x > 0 \implies (x - 1 \neq 0 \wedge \frac{1}{x-1} > 0) \tag{1}$$

A modern automated theorem prover (e.g. [53, 54]) will quickly establish this condition does not hold. At this point, the verifying compiler should report a helpful error message. Unfortunately, during the weakest precondition transform, information about where exactly the error arose was lost. To identify where the error occurred, there are two intrinsic questions we need to answer: *where exactly in the program code does the error arise?* and, *which execution path(s) give rise to the error?* The $wp(\mathtt{s}, \mathtt{q})$ transform fails to answer both because it generates a single verification condition for the entire function which is either shown to hold, or not [55, 56].

One strategy for resolving this issue is to embed attributes in the verification condition identifying where in the original source program particular components originated [7, 57, 58, 55, 59]. Unfortunately, this requires specific support from the automated theorem prover (which is not always available). Another approach is to try and split the generated weakest precondition into those components corresponding to different execution paths (for example, there are two clear disjuncts in our example above). In general, this is challenging because the generated conditions can be arbitrarily complex [57].

Strongest preconditions. Instead of operating in a backwards direction, our experience suggests it is inherently more practical to generate verification conditions in a forwards direction (and there is anecdotal evidence to support this [47]). Recall that this corresponds to generating strongest postconditions, rather than weakest preconditions. The key advantage is that verification conditions can be emitted at the specific points where failures may occur.

In the above example, there are two potential failures: firstly, `1/(x-1)` should not cause a division-by-zero; secondly, the postcondition `$ > 0` must be met. A forward propagating verification condition generator can generate separate conditions for each potential failure. For example, it can emit the following verification conditions:

$$\mathtt{x} > 0 \implies \mathtt{x} - 1 \neq 0 \tag{2}$$

$$\mathtt{x} > 0 \implies \frac{1}{\mathtt{x} - 1} > 0 \tag{3}$$

Each of these can be associated with the specific program point where it originated and, in the case it cannot be shown, an error can be reported at that point. For example, since the first verification condition above does not hold, an error can be reported for the statement $\mathtt{x} = \mathtt{1}/(\mathtt{x} - \mathtt{1})$. When generating verification conditions based on $wp(\mathtt{s}, \mathtt{q})$, it is hard to report errors at the specific point they arise because, at each point, only the weakest precondition for *subsequent* statements is known.

6 Related Work

Hoare provided the foundation for formalising work in this area with his seminal paper introducing *Hoare Logic* [21]. This provides a framework for proving that a sequence of statements meets its postcondition given its precondition. Unfortunately Hoare logic does not tell us how to *construct* such a proof; rather, it gives a mechanism for *checking* a proof is correct. Therefore, to actually verify a program is correct, we need to construct proofs which satisfy the rules of Hoare logic.

The most common way to automate the process of verifying a program is with a verification condition generator. As discussed in §4.2, such algorithms propagate information in either a forwards or backwards direction. However, the rules of Hoare logic lend themselves more naturally to the latter [47]. Perhaps for this reason, many tools choose to use the weakest precondition transformer. For example, the widely acclaimed ESC/Java tool computes weakest preconditions [7], as does the Why platform [60], Spec# [61], LOOP [62], JACK [59] and SnuggleBug [63]. This is surprising given that it leads to fewer verification conditions and, hence, makes it harder to generate useful error messages (recall our discusion from §4.2). To workaround this, Burdy *et al.* embed path information in verification conditions to improve error reporting [59]. A similar approach is taken in ESC/Java, but requires support from the underlying automated theorem prover [54]. Denney and Fischer extend Hoare logic to formalise the embedding of information within verification conditions [57]. Again, their objective is to provide useful error messages.

A common technique for generating verification conditions is to transform the input program into *passive form* [6, 7, 61, 64]. Here, the control-flow graph of each function is converted using standard techniques into a reducible (albeit potentially larger) graph. This is then further reduced by eliminating loops to leave an acyclic graph, before a final transformation into *Static Single Assignment form (SSA)* [65, 66]. The main advantage is that, after this transformation, generating verification conditions becomes straightforward. Furthermore, the technique works well for unstructured control flow and can be tweaked to produced compact verification conditions [67, 68, 69, 70]. However, an important issue (from the perspective of this paper) is that this makes it somewhat challenging to formalise the whole process.

Dijkstra's *Guarded Command Language* provides an alternative approach to the generation of verification conditions [22]. In this case, the language is far removed from the simple imperative language of Hoare logic and, for example, contains only the sequence and non-deterministic choice constructs for handling control-flow. There is rich history of using guarded commands as an intermediate language for verification, which began with the ECS/Modula-3 tool [6]. This was continued in ESC/Java and, during the later development of Spec#, a richer version (called Boogie) was developed [11]. Such tools use guarded commands as a way to represent programs that are in passive form (discussed above) in a human-readable manner. As these programs are acyclic, the looping constructs of Dykstra's original language are typically ignored.

Finally, it is worth noting that Frade and Pinto provide an excellent survey of verification condition generation for simple WHILE programs [47]. They primarily focus on Hoare Logic and various extensions, but also explore Dijkstra's Guarded Command Language. They consider an extended version of Hoare's While Language which includes user-provided loop invariants. They also present an algorithm for generating verification conditions based on the weakest precondition transformer.

7 Conclusion

In this paper, we have reflected on our experiences using the Whiley verifying compiler. In particular, we have identified a number of practical considerations for any verifying compiler which are not immediately obvious from the underlying theoretical foundations.

Acknowledgements. This work is supported by the Marsden Fund, administered by the Royal Society of New Zealand.

References

[1] C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *JACM*, 50(1):63–69, 2003.

[2] S. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.

[3] L. Peter Deutsch. *An interactive program verifier*. Ph.d., 1973.

[4] D. I. Good. Mechanical proofs about computer programs. In *Mathematical logic and programming languages*, pages 55–75, 1985.

[5] David C. Luckham, Steven M. German, Friedrich W. von Henke, Richard A. Karp, P. W. Milne, Derek C. Oppen, Wolfgang Polak, and William L. Scherlis. Stanford pascal verifier user manual. Technical Report CS-TR-79-731, Stanford University, Department of Computer Science, 1979.

[6] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 1998.

[7] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245. ACM Press, 2002.

[8] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, March 2005.

[9] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The spec# programming system: An overview. Technical report, Microsoft Research, 2004.

[10] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[11] 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 *Proc. FMCO*, pages 364–387, 2006.

[12] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. LPAR*, LNCS. Springer-Verlag, 2010.

[13] K. Rustan M. Leino. Developing verified programs with dafny. In *Proc. VSTTE*, volume 7152 of *LNCS*, pages 82–82. Springer-Verlag, 2012.

[14] The whiley programming language, http://whiley.org.

[15] D. J. Pearce and L. Groves. Whiley: a platform for research in software verification. In *Proc. SLE*, page (to appear), 2013.

[16] D. Pearce and J. Noble. Implementing a language with flow-sensitive and structural typing on the JVM. *Electronic Notes in Theoretical Computer Science*, 279(1):47–59, 2011.

[17] D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Proc. VMCAI*, pages 335–354, 2013.

[18] D. J. Pearce. A calculus for constraint-based flow typing. In *Proc. Workshop on Formal Techniques for Java-like Programs*, page Article 7, 2013.

[19] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. A. Brady. Deciding bit-vector arithmetic with abjc10straction. In *Proc. TACAS*, volume 4424 of *LNCS*, pages 358–372. Springer-Verlag, 2007.

[20] R. W. Floyd. Assigning meaning to programs. In *Proc. AMS*, volume 19, pages 19–31. American Mathematical Society, 1967.

[21] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12, 1969.

[22] Edsger W. Dijkstra. Guarded commands, nondeterminancy and formal derivation of programs. *CACM*, 18:453–457, 1975.

[23] David J. Pearce. JPure: a modular purity system for Java. In *Proc. CC*, pages 104–123, 2011.

[24] Atanas Rountev. Precise identification of side-effect-free methods in java. In *Proc. ICSM*, pages 82–91. IEEE Computer Society, 2004.

[25] A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. In *Proc. VMCAI*, pages 199–215, 2005.

[26] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. *SIGSOFT Softw. Eng. Notes*, 27(4):1–11, 2002.

[27] Alan Mycroft. Programming language design and analysis motivated by hardware evolution. In *Proc. SAS*, pages 18–13. Springer-Verlag, 2007.

[28] Nurudeen Lameed and Laurie J. Hendren. Staged static techniques to efficiently implement array copy semantics in a MATLAB JIT compiler. In *Proc. CC*, pages 22–41, 2011.

[29] Natarajan Shankar. Static analysis for safe destructive updates in a functional language. In *In Proc. LOPSTR*, pages 1–24, 2001.

[30] Martin Odersky. How to make destructive updates less destructive. In *Proc. POPL*, pages 25–36, 1991.

[31] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proc. ICFP*, pages 117–128, 2010.

[32] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *Proc. ESOP*, pages 256–275, 2011.

[33] Franco Barbanera and Mariangiola Dezani-Cian Caglini. Intersection and union types. In *In Proc. TACS*, pages 651–674, 1991.

[34] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. *Journal of Object Technology*, 6(2), 2007.

[35] Tony Hoare. Null references: The billion dollar mistake, presentation at qcon, 2009.

[36] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[37] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. OOPSLA*, pages 302–312. ACM Press, 2003.

[38] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6(9):455–475, 2007.

[39] Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proc. ECOOP*, pages 227–247. Springer, 2007.

[40] Chris Male, David J. Pearce, Alex Potanin, and Constantine Dymnikov. Java bytecode verification for @NonNull types. In *Proc. CC*, pages 229–244, 2008.

[41] Laurent Hubert. A non-null annotation inferencer for java bytecode. In *Proc. PASTE*, pages 36–42. ACM, 2008.

[42] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *JACM*, 55(4):19:1–19:64, 2008.

[43] ISO/IEC. international standard ISO/IEC 9899, programming languages — C, 1990.

[44] Patrice Chalin and Frédéric Rioux. JML runtime assertion checking: Improved error reporting and efficiency using strong validity. In *Proc. FM*, volume 5014 of *LNCS*, pages 246–261. Springer-Verlag, 2008.

[45] Stan Jefferson and Samuel N. Kamin. Executable specifications with quantifiers in the fase system. In *Proc. POPL*, pages 318–326. ACM, 1986.

[46] N. E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, September 1992.

[47] Maria João Frade and Jorge Sousa Pinto. Verification conditions for source-level imperative programs. 5(3):252–277, 2011.

[48] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[49] Mike Gordon and Hélène Collavizza. Forward with Hoare. In *Reflections on the Work of C.A.R. Hoare*, History of Computing, pages 101–121. Springer-Verlag, 2010.

[50] R. Chadha and D. A. Plaisted. On the mechanical derivation of loop invariants. *Journal of Symbolic Computation*, 15(5 & 6):705–744, 1993.

[51] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *Proc. APLAS*, volume 3780 of *LNCS*, pages 119–134. Springer-Verlag, 2005.

[52] Carlo A. Furia and Bertrand Meyer. Inferring loop invariants using postconditions. *CoRR*, abs/0909.0884, 2009.

[53] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, pages 337–340, 2008.

[54] Detlefs, Nelson, and Saxe. Simplify: A theorem prover for program checking. *JACM*, 52, 2005.

[55] K. Rustan M. Leino, Todd D. Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1-3):209–226, 2005.

[56] Ivan Jager and David Brumley. Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, Carnegie Mellon University, 2010.

[57] Ewen Denney and Bernd Fischer. Explaining verification conditions. In *Proceedings of the Conference on Algebraic Methodology and Software Technology (AMAST)*, volume 5140 of *LNCS*, pages 145–159. Springer, 2008.

[58] Ranan Fraer. Tracing the origins of verification conditions. In *Proc. AMAST*, volume 1101 of *LNCS*, pages 241–255. Springer-Verlag, 1996.

[59] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In *Proc. FME*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.

[60] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of CAV*, volume 4590 of *LNCS*, pages 173–177, 2007.

[61] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proc. PASTE*, pages 82–87. ACM Press, 2005.

[62] B. Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *JLAP*, 58(1–2):61–88, 2004.

[63] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Proc. PLDI*, pages 363–374. ACM Press, 2009.

[64] Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest postcondition of unstructured programs. In *Proc. Workshop on Formal Techniques for Java-like Programs*, pages 6:1–6:7. ACM Press, 2009.

[65] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proc. POPL*, pages 25–35. ACM Press, 1989.

[66] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, 1991.

[67] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proc. POPL*, pages 193–205. ACM Press, 2001.

[68] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93, 2005.

[69] Domagoj Babic and Alan J. Hu. Structural abstraction of software verification conditions. In *Proceedings of CAV*, volume 4590 of *LNCS*, pages 366–378. Springer-Verlag, 2007.

[70] Domagoj Babic and Alan J. Hu. Exploiting shared structure in software verification conditions. In *Haifa Verification Conference*, volume 4899 of *LNCS*, pages 169–184. Springer, 2007.