The Whiley Language Specification

David J. Pearce
School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
djp@ecs.vuw.ac.nz

January 13, 2014

Contents

Chapter 1

Introduction

This document provides a specification of the *Whiley Programming Language*. Whiley is a hybrid imperative and functional programming language designed to produce programs with fewer errors that those developed by more convention means. Whiley allows explicit specifications to be given for functions, methods and data structures, and employs a *verifying compiler* to check whether programs meet their specifications. As such, Whiley is ideally suited for use in *safety critical system*s. However, there are many benefits to be gained from using Whiley in a general setting (e.g. improved documentation, maintainability, reliability, etc). Finally, this document is *not* intended as a general introduction to the language, and the reader is referred to alternative documents for learning the language [**?**].

1.1 Background

Reliability of large software systems is a difficult problem facing software engineering, where subtle errors can have disastrous consequences. Infamous examples include: the Therac-25 disaster where a computer-operated X-ray machine gave lethal doses to patients [1]; the 1988 worm which reeked havoc on the internet by exploiting a buffer overrun [2]; the 1991 Patriot missile failure where a rounding error resulted in the missile catastrophically hitting a barracks [3]; and, the Ariane 5 rocket which exploded shortly after launch because of an integer overflow, costing the ESA an estimated $500 million [4].

The most widely used and accepted approach to improving software reliability is through extensive testing and manual code inspection. Whilst this does increase confidence, it cannot guarantee the absence of errors — which is particularly problematic in a safety-critical setting. Another successful approach is to prove the correctness of *models of software*, rather than of the software itself. For example, model checkers (e.g [5, 6, 7]) and SAT solvers (e.g. [8, 9]) have proved highly effective at checking correctness properties of finite models of software systems, including microprocessor designs [10, 11], flight-control systems [12, 13], network protocols [14, 15] and spaceflight-control systems [16]. Some model checkers (e.g. CBMC [17], Java Pathfinder [16], BLAST [18], SLAM [19]) can also be applied directly on the program code although, in such cases, either significant abstraction is performed (hence, reducing the scope) or scalability is sacrificed.

Prof. Sir Tony Hoare (ACM Turing Award Winner, FRS) proposed the creation of a *verifying compiler* as a grand challenge for computer science [20]. A verifying compiler "*uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles.*" There have been numerous attempts to construct a verifying compiler system, although none has yet made it into the mainstream. Early examples include that of King [21], Deutsch [22], the Gypsy Verification Environment [23] and the Stanford Pascal Verifier [24]. More recently, the Extended Static Checker for Modula-3 [25] which became the Extended Static Checker for Java (ESC/Java) — a widely acclaimed and influential work [26]. Building on this success was JML and its associated tooling which provided a standard notation for specifying functions in Java [27]. Finally, Microsoft developed the Spec# system which is built on top of C# [28].

Both ESC/Java and Spec# build on existing object-oriented languages (i.e. Java and C#) but, as a result, suffer numerous limitations. The problem is that such languages were not designed for use with verifying compilers. Ireland, in his survey on the history of verifying compilers, noted the following [29]:

> *"The choice of programming language(s) targeted by the verifying compiler will have a significant effect on the chances of success."*

Likewise, a report on future directions in verifying compilers, put together by several researchers in this area, makes a similar comment [30]:

> *"Programming language design can reduce the cost of specification and verification by keeping the language simple, by automating more of the work, and by eliminating common errors."*

1.2 Goals

The Whiley Programming Language has been designed from scratch in conjunction with a verifying compiler. The intention of this is to provide an open framework for research in automated software verification. The initial goal is to automatically eliminate common errors, such as *null dereferences*, *array-out-of-bounds*, *divide-by-zero* and more. In the future, the intention is to consider more complex issues, such as termination, proof-carrying code and user-supplied proofs.

1.3 History

Chapter 2

Lexical Structure

2.1 Indentation

2.2 Blocks

2.3 Whitespace

2.4 Identifiers

Chapter 3

Source Files

Whiley programs are split across one or more *source file*s which are compiled into *WyIL file*s prior to execution. Source files contain declarations which describe the functions, methods, data types and constants which form the program. Source files are grouped together into coherent units called *package*s.

3.1 Compilation Units

3.2 Packages & Imports

3.3 Declarations

Camel case

3.3.1 Access Control

3.3.2 Type Declarations

A *type declaration* declares a named type within a Whiley source file. The declaration may refer to named types in this or other source filess and may also *recursively* refer to itself (either directly or indirectly).

```
TypeDecl  ::=  type Ident is TypePattern [ where Expr ]
```

The optional **where** clause defines a *boolean expression* which holds for any instance of this type. This is often referred to as the type *invariant* or *constraint*. Variables declared within the *type pattern* may be referred to within the optional **where** clause.

Examples. Some simple examples illustrating type declarations are:

```
// Define a simple point type
type Point is { int x, int y }

// Define the type of natural numbers
type nat is (int x) where x >= 0
```

The first declaration defines an unconstrained record type named `Point`, whilst the second defines a constrained integer type `nat`.

Notes. A convention is that type declarations for *records* or *unions of records* begin with an upper case character (e.g. `Point` above). All other type declarations begin with lower case. This reflects the fact that records are most commonly used to describe objects in the domain.

3.3.3 Constant Declarations

A *constant declaration* declares a named constant within a Whiley source file. The declaration may refer to named constants in this or other source filess, although it may not refer to itself (either directly or indirectly).

```
ConstantDecl  ::=  constant Ident is Expr
```

The given *constant expression* is evaluated at *compile time* and must produce a constant value. This prohibits the use of function or method calls within the constant expression. However, general operators (e.g. for arithmetic) are permitted.

Examples. Some example to illustrate constant declarations are:

```
// Define the well-known mathematical constant to 10 decimal places.
constant PI is 3.141592654

// Define a constant expression which is twice PI
constant TWO_PI is PI * 2.0
```

The first declaration defines the constant `PI` to have the **`real`** value `3.141592654`. The second declaration illustrates a more interesting constant expression which is evaluated to `6.283185308` at compile time.

Notes. A convention is that constants are named in upper case with underscores separating words (i.e. as in `TWO_PI` above).

3.3.4 Function Declarations

A *function declaration* defines a function within a Whiley source file. Functions are *pure* and may not have side-effects. This means they are guaranteed to always return the same result given the same arguments, and are permitted within specifications (i.e. in type invariants, *loop invariant*s, and function/method *precondition*s or *postcondition*s). Functions may call other functions, but may not call other methods. They also may not allocate memory on the heap and/or instigate concurrent computation.

```
FunctionDecl  ::=  function Ident TypePattern => TypePattern (
                     throws Type | requires Expr | ensures Expr
                   )* : Block
```

The first type pattern (i.e. before "=>") is referred to as the *parameter*, whilst the second is referred to as the *return*. There are three kinds of optional clause which follow:

- **Throws clause**. This defines the exceptions which may be thrown by this function. Multiple clauses may be given, and these are taken together as a union. Furthermore, the convention is to specify the throws clause before the others.

- **Requires clause(s)**. These define constraints on the permissible values of the parameters on entry to the function or method, and are often collectively referred to as the precondition. These expressions may refer to any variables declared within the parameter type pattern. Multiple clauses may be given, and these are taken together as a conjunction. Furthermore, the convention is to specify the requires clause(s) before any ensure(s) clauses.
- **Ensures clause(s)**. These define constraints on the permissible values of the the function or method's return value, and are often collectively referred to as the postcondition. These expressions may refer to any variables declared within either the parameter or return type pattern. Multiple clauses may be given, and these are taken together as a conjunction. Furthermore, the convention is to specify the requires clause(s) after the others.

Examples. The following function declaration provides a small example to illustrate:

```
function max(int x, int y) => (int z)
// return must be greater than either parameter
ensures x <= z && y <= z
// return must equal one of the parmaeters
ensures x == z || y == z:
    // implementation
    if x > y:
        return x
    else:
        return y
```

This defines the specification and implementation of the well-known `max()` function which returns the largest of its parameters. This does not throw any exceptions, and does not enforce any preconditions on its parameters.

3.3.5 Method Declarations

A *method declaration* defines a method within a Whiley source file. Methods are *impure* and may have side-effects. Thus, they cannot be used within specifications (i.e. in type invariants, loop invariants, and function/method preconditions or postconditions). However, unlike functions, they methods call other functions and/or methods (including `native` methods). They may also allocate memory on the heap, and/or instigate concurrent computation.

```
MethodDecl   ::=   method Ident TypePattern => TypePattern (
                    throws Type | requires Expr | ensures Expr
                   )* : Block
```

The first type pattern (i.e. before "=>") is referred to as the *parameter*, whilst the second is referred to as the *return*. The three optional clauses are defined identically as for functions above.

Examples. The following method declaration provides a small example to illustrate:

```
// Define the well-known concept of a linked list
type LinkedList is null | { &LinkedList next, int data }

// Define a method which inserts a new item onto the end of the list
method insertAfter(&LinkedList list, int item):
    if *list is null:
        // reached the end of the list, so allocate new node
        ⋆list = new { next: null, data: item }
    else:
        // continue traversing the list
        insertAfter(list→next, item)
```

Chapter 4

Types

4.1 Overview

Discuss syntactic versus semantic types. Also, need to consider constrained types as well as type patterns.

```
Type  ::=
        |   TermType
        |   UnionType
        |   IntersectionType
```

```
TermType  ::=
            |   PrimitiveType
            |   TupleType
            |   RecordType
            |   ReferenceType
            |   NominalType
            |   CollectionType
            |   NegationType
            |   FunctionType
            |   MethodType
```

4.2 Primitives

```
PrimitiveType  ::=
                 |   AnyType
                 |   VoidType
                 |   NullType
                 |   BoolType
                 |   ByteType
                 |   CharType
                 |   IntType
                 |   RealType
```

4.2.1 Any Type

The type **any** represents the type whose variables may hold any possible value. Thus, **any** is the *top type* (i.e. ⊤) in the lattice of types and, hence, is the supertype of all other types.

```
AnyType ::= any
```

Examples.

Semantics.

4.2.2 Void Type

The **void** type represents the type whose variables cannot exist (i.e. because they cannot hold any possible value). Thus, **void** is the *bottom type* (i.e. ⊥) in the lattice of types and, hence, is the *subtype* of all other types. Void is used to represent the return type of a method which does not return anything. Furthermore, it is also used to represent the element type of an empty list of set.

```
VoidType ::= void
```

Examples. The following example illustrates several uses of the **void** type:

```
// Attempt to update first element
method update1st(&[int] list, int value) => void:
    // First, check whether list is empty or not
    if *list != [void]:
        // Then, update 1st element
        (*list)[0] = x
    // done
```

Semantics.

4.2.3 Null Type

The null type is a special type which should be used to show the absence of something. It is distinct from void, since variables can hold the special **null** value (where as there is no special ”**void**” value).

```
NullType ::= null
```

Examples.

Semantics.

Notes. With all of the problems surrounding **`null`** and `NullPointerException`s in languages like Java and C, it may seem that this type should be avoided. However, it remains a very useful abstraction around (e.g. for terminating recursive types) and, in Whiley, is treated in a completely safe manner (unlike e.g. Java).

4.2.4 Bool Type

Represents the set of boolean values (i.e. `true` and `false`).

```
BoolType  ::=  bool
```

Examples.

Semantics.

Notes.

4.2.5 Byte Type

Represents a sequence of 8 bits.

```
ByteType  ::=  byte
```

Examples.

Semantics.

Notes. Unlike for many languages, there is no representation associated with a byte. For example, to extract an integer value from a byte, it must be explicitly decoded according to some representation (e.g. two's compliment) using an auxillary function (e.g. `Byte.toInt()`).

4.2.6 Char Type

Represents a arbitrary unicode character.

```
CharType  ::=  char
```

Examples.

Semantics.

Notes.

4.2.7 Int Type

Represents the set of (unbound) integer values.

$$\texttt{IntType} \quad ::= \quad \boxed{\texttt{int}}$$

Examples.

Semantics.

Notes. Since integer types in Whiley are unbounded, there is no equivalent to Java's `MIN_VALUE` and `MAX_VALUE` for **`int`** types.

4.2.8 Real Type

Represents the set of (unbound) rational numbers.

$$\texttt{RealType} \quad ::= \quad \boxed{\texttt{real}}$$

Examples.

Semantics.

Notes.

4.3 Tuple Types

A tuple type describes a compound type made up of two or more subcomponents. It is similar to a record, except that fields are effectively anonymous.

$$\texttt{TupleType} \quad ::= \quad \boxed{\texttt{(}}\ \texttt{Type}\ \Big(\boxed{\texttt{,}}\ \texttt{Type}\ \Big)^{+}\ \boxed{\texttt{)}}$$

Examples.

Semantics.

Notes.

4.4 Record Types

A record is made up of a number of fields, each of which has a unique name. Each field has a corresponding type. One can think of a record as a special kind of ”fixed” map (i.e. where we know exactly which entries we have).

```
RecordType  ::=  { Type Ident ( , Type Ident )* [ , ... ] }
```

Examples.

Semantics.

Notes. Syntax for functions? Open versus closed records?

4.5 Reference Types

Represents a reference to an object in Whiley.

```
ReferenceType  ::=  & Type
```

Examples.

Semantics.

Notes.

4.6 Nominal Types

The existential type represents the an unknown type, defined at a given position.

```
NominalType  ::=  Ident
```

Examples.

Semantics.

Notes.

4.7 Collection Types

4.7.1 Set Type

A set type describes set values whose elements are subtypes of the element type. For example, `{1,2,3}` is an instance of set type `{int}`; however, `{1.345}` is not.

```
SetType  ::=  { Type }
```

Examples.

Semantics.

Notes.

4.7.2 Map Type

A map represents a one-many mapping from variables of one type to variables of another type. For example, the map type `{int=>real}` represents a map from integers to real values. A valid instance of this type might be `{1=>1.2,2=>3.0}`.

```
MapType  ::=  { Type => Type }
```

Examples.

Semantics.

Notes.

4.7.3 List Type

A list type describes list values whose elements are subtypes of the element type. For example, `[1,2,3]` is an instance of list type `[int]`; however, `[1.345]` is not.

```
ListType  ::=  [ Type ]
```

Examples.

Semantics.

Notes.

4.8 Function Types

$$\texttt{FunctionType} \quad ::= \quad \boxed{\texttt{function}}\ \boxed{\texttt{(}}\ \big[\ \texttt{Type}\ \big(\ \boxed{\texttt{,}}\ \texttt{Type}\ \big)^*\ \big]\ \boxed{\texttt{)}}\ \boxed{\texttt{=>}}\ \texttt{Type}$$

Description.

Examples.

Semantics.

Notes.

4.9 Method Types

$$\texttt{MethodType} \quad ::= \quad \boxed{\texttt{method}}\ \boxed{\texttt{(}}\ \big[\ \texttt{Type}\ \big(\ \boxed{\texttt{,}}\ \texttt{Type}\ \big)^*\ \big]\ \boxed{\texttt{)}}\ \boxed{\texttt{=>}}\ \texttt{Type}$$

Description.

Examples.

Semantics.

Notes.

4.10 Union Types

A union type represents a type whose variables may hold values from any of its ”bounds”. For example, the union type **`null|int`** indicates a variable can either hold an integer value, or **`null`**.

$$\texttt{UnionType} \quad ::= \quad \texttt{IntersectionType}\ \big(\ \boxed{\texttt{|}}\ \texttt{IntersectionType}\ \big)^+$$

Examples.

Semantics.

Notes. There must be at least two bounds for a union type to make sense.

4.11 Intersection Types

$$\texttt{IntersectionType} \quad ::= \quad \texttt{TermType}\ \big(\ \boxed{\texttt{\&}}\ \texttt{TermType}\ \big)^+$$

Description.

Examples.

Semantics.

Notes.

4.12 Negation Types

NegationType ::= $\boxed{!}$ `Type`

Description. A negation type represents a type which accepts values *not* in a given type.

Examples.

Semantics.

Notes.

4.13 Abstract Types

4.13.1 Recursive Types

4.13.2 Effective Tuples

4.13.3 Effective Records

4.13.4 Effective Collections

4.14 Subtyping Algorithms

Discussion of soundness and completeness.

Chapter 5

Statements

5.1 Assert Statement

Represents an *assert statement* of the form “**assert** e”, where e is a boolean expression.

```
AssertStmt   ::=   assert Expr
```

Examples. The following illustrates:

```
function abs(int x) => int:
    if x < 0:
        x = -x
    assert x >= 0
    return x
```

Notes. Assertions are either *statically checked* by the verifier, or turned into *runtime checks*.

5.2 Assignment Statement

Represents an *assignment statement* of the form lhs = rhs. Here, the rhs is any expression, whilst the lhs must be an LVal — that is, an expression permitted on the left-side of an assignment.

```
AssignStmt   ::=   LVal = Expr
```

Examples. The following illustrates different possible assignment statements:

```
x = y        // variable assignment
x.f = y      // field assignment
x[i] = y     // list assignment
x[i].f = y   // compound assignment
```

The last assignment here illustrates that the left-hand side of an assignment can be arbitrarily complex, involving nested assignments into lists and records.

Semantics.

Notes.

5.3 Assume Statement

Represents an *assume statement* of the form “`assume e`”, where `e` is a boolean expression.

```
AssumeStmt   ::=   assume Expr
```

Examples. The following illustrates a simple function which uses an `assume` statement to meet its postcondition:

```
function abs(int x) => int:
    assume x >= 0
    return x
```

Notes. Assumptions are *assumed* by the verifier and, since this may be unsound, are always turned into *runtime checks*.

5.4 Return Statement

Represents a *return statement* with an optional expression is referred to as the *return value*.

```
ReturnStmt   ::=   return [Expr]
```

Examples. The following illustrates a simple function which returns the increment of its parameter `x`:

```
function f(int x) => int:
    return x + 1
```

Here, we see a simple **`return`** statement which returns an **`int`** value.

Notes. The returned expression (if there is one) must begin on the same line as the return statement itself.

5.5 Throw Statement

```
ThrowStmt   ::=   throw Expr
```

Description.

Examples.

Notes.

5.6 Variable Declarations

Represents a *variable declaration* which has an optional expression assignment referred to as an *variable initialiser*. If an initialiser is given, then this will be evaluated and assigned to the variable when the declaration is executed.

$$\texttt{VarDecl} \quad ::= \quad \texttt{Type}\ \texttt{Ident}\ \left[\, \boxed{\texttt{=}}\ \texttt{Expr} \,\right]$$

Examples. Some example variable declarations are:

```
int x
int y = 1
int z = x + y
```

Notes.

5.7 If Statement

Represents a classical `if` statement which supports chaining and an optional `else` branch. The expression(s) are referred to as *conditions* and must be boolean expressions. The first block is referred to as the *true branch*, whilst the optional `else` block is referred to as the *false branch*.

$$\begin{aligned} \texttt{IfStmt}^{\ell} \quad ::= \quad & \boxed{\texttt{if}}\ \texttt{Expr}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma}\ \left(\, \boxed{\texttt{else}}\ \boxed{\texttt{if}}\ \texttt{Expr}\ \boxed{\texttt{:}}\ \texttt{Block}^{\omega_i} \,\right)^{*} \\ & \left[\, \boxed{\texttt{else}}\ \boxed{\texttt{:}}\ \texttt{Block}^{\phi} \,\right] \\ & \qquad\qquad\qquad (\text{where } \ell < \gamma \text{ and } \forall i.\ell < \omega_i \text{ and } \ell < \phi) \end{aligned}$$

Examples. The following illustrates:

```
function max(int x, int y) => int:
    if(x > y):
        return x
    else if(x == y):
        return 0
    else:
        return y
```

Notes.

5.8 While Statement

Represents a while statement with optional **`where`** clause(s) commonly referred to as loop invariants.

$$\texttt{WhileStmt}^{\ell} \quad ::= \quad \boxed{\texttt{while}}\ \texttt{Expr}\ \Big(\ \boxed{\texttt{where}}\ \texttt{Expr}\ \Big)^{*}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma}$$

$$(\text{where } \ell < \gamma)$$

Examples. As an example:

```
function sum([int] xs) => int:
  int r = 0
  int i = 0
  while i < |xs| where i >= 0:
    r = r + xs[i]
    i = i + 1
  return r
```

Notes. When multiple **`where`** clauses are given, these are combined using a conjunction. The combined invariant defines a condition which must be true on every iteration of the loop.

5.9 Do/While Statement

$$\texttt{DoWhileStmt}^{\ell} \quad ::= \quad \boxed{\texttt{do}}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma}\ \boxed{\texttt{while}}\ \texttt{Expr}\ \Big(\ \boxed{\texttt{where}}\ \texttt{Expr}\ \Big)^{*}$$

$$(\text{where } \ell < \gamma)$$

Description.

Examples.

Notes.

5.10 For Statement

$$\texttt{ForStmt}^{\ell} \quad ::= \quad \boxed{\texttt{for}}\ \texttt{VarPattern}\ \boxed{\texttt{in}}\ \texttt{Expr}\ \Big(\ \boxed{\texttt{where}}\ \texttt{Expr}\ \Big)^{*}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma}$$

$$(\text{where } \ell < \gamma)$$

Description.

Examples.

Notes.

5.11 Switch Statement

```
SwitchStmt  ::=
```

Description.

Examples.

Notes.

5.12 Try/Catch Statement

```
TryCatchStmt  ::=
```

Description.

Examples.

Notes.

Chapter 6

Expressions

Expression blah blah.

6.1 Tuple Expressions

$$\texttt{TupleExpr} \quad ::= \quad \texttt{UnitExpr}\,(\,\boxed{\texttt{,}}\,\texttt{UnitExpr}\,)^{+}$$

Description.

Examples.

Notes.

6.2 Unit Expressions

$$\texttt{UnitExpr} \quad ::= \quad \texttt{LogicalExpr}$$

Description.

Examples.

Notes.

6.3 Logical Expressions

```
   LogicalExpr  ::=  LogicalOrExpr

  LogicalOrExpr  ::=  LogicalAndExpr
                  |   LogicalOrExpr || LogicalAndExpr

LogicalAndExpr  ::=  BitwiseExpr
                  |   LogicalAndExpr && BitwiseExpr
```

Description.

Examples.

Notes.

6.4 Bitwise Expressions

```
   BitwiseExpr  ::=  BitwiseOrExpr

  BitwiseOrExpr  ::=  BitwiseXorExpr
                  |   BitwiseOrExpr | BitwiseXorExpr

BitwiseXorExpr  ::=  BitwiseAndExpr
                  |   BitwiseXorExpr ^ BitwiseAndExpr

BitwiseAndExpr  ::=  ConditionExpr
                  |   BitwiseAndExpr && ConditionExpr
```

Description.

Examples.

Notes.

6.5 Condition Expressions

```
ConditionExpr  ::=
```

Description.

Examples.

Notes.

6.6 Quantifier Expressions

```
QuantExpr  ::=  ( no | some | all ) {
                 Ident in Expr ( , Ident in Expr )+ | LogicalExpr
                }
```

Description.

Examples.

Notes.

6.7 Append Expressions

```
AppendExpr  ::=  RangeExpr ( ++ RangeExpr )*
```

Description.

Examples.

Notes.

6.8 Range Expressions

```
RangeExpr  ::=  ShiftExpr [ .. ShiftExpr ]
```

Description.

Examples.

Notes.

6.9 Shift Expressions

```
ShiftExpr  ::=  AdditiveExpr [ ( << | >> ) AdditiveExpr ]
```

Description.

Examples.

Notes.

6.10 Additive/Multiplicative Expressions

```
      AdditiveExpr  ::=
MultiplicativeExpr  ::=
```

Description.

Examples.

Notes.

6.11 Access Expressions

```
AccessExpr  ::=
```

Description.

Examples.

Notes.

6.12 Term Expressions

```
TermExpr  ::=
```

Description.

Examples.

Notes.

Glossary

boolean expression An expression which evaluates to a value of type `bool`. 6, 18–20, 27

expression A combination of constants, variables and operators that, when evaluated, produce a single value. Expressions in certain circumstances may have side effects. 23, 27

loop invariant A boolean expression which must hold on every iteration of a loop. 7, 8, 21

package A unit of hierarchical organisation within the Whiley namespace.. 6

postcondition A logical condition over the parameters and returns of a function or method which must be true immediately after execution of that function or method.. 7, 8

precondition A logical condition over the parameters of a function or method which must be true immediately prior to execution of that function or method.. 7, 8

safety critical system A system which operates in a high-risk setting where failure can lead to loss of life, injury, significant damage or environmental harm. 3

source file A file in which source code is located. Source files for the Whiley programming language have the extension `.whiley`. In Whiley, source files must be compiled into a binary form before they can be executed.. 6–8, 27

type An descriptor for a set of values, typically used to determine the set of values a given variable or expression may hold. 27

variable declaration A statement which declares one or more variable(s) for use in a given scope. Each variable is given a *type* which limits the possible values it may hold, and may not already be declared in an enclosing scope. 20, 27

variable initialiser An optional expression used to initialise variable(s) declared as part of a variable declaration. 20

verifying compiler A compilers which employs automated mathematical and logical reasoning to check the correctness of the programs that it compiles. 3

WyIL file A compiled (i.e. binary) form of a Whiley source file. 6

Bibliography

[1] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[2] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the internet virus of November 1988. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 326–343, 1989.

[3] Software problem led to system failure at dhahran, saudi arabia, gao report #b-247094, 1992.

[4] Ariane 5: Flight 501 failure. report by the enquiry board. Technical report, European Space Agency, 1996.

[5] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[6] Gerard J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–95, 1997.

[7] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. POPL*, pages 1–3, 2002.

[8] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of DAC*, 2001.

[9] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient SAT solver. In *Proc. of SAT*, pages 360–375. Springer, 2004.

[10] Miroslav N. Velev and Randal E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, 2003.

[11] Thomas Schubert. High level formal verification of next-generation microprocessors. In *Proc. of DAC*, pages 1–6. ACM, 2003.

[12] P. R. Gluck and G. J. Holzmann. Using SPIN model checking for flight software verification. In *IEEE Aerospace Conference*, pages 105–113. IEEE Computer Society Press, 2002.

[13] Yunja Choi. Model checking flight guidance systems: from synchrony to asynchrony. *Electronic Notes in Computer Science*, 133:61–79, 2005.

[14] Dominique Bolignano. Integrating proof-based and model-checking techniques for the formal verification of cryptographic protocols. In *Proc. of CAV*, pages 77–87. Springer, 1998.

[15] A. Armando and L. Compagna. Abstraction-driven SAT-based analysis of security protocols. In *Proc. of SAT*, pages 257–271. Springer, 2003.

[16] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[17] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer-Verlag, 2004.

[18] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In *Proceedings of the Workshop on Model Checking Software*, pages 235–239. Springer-Verlag, 2003.

[19] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. PLDI*, pages 203–213. ACM Press, 2001.

[20] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

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

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

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

[24] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford pascal verifier user manual. Technical Report CS-TR-79-731, Stanford University, Department of Computer Science, 1979.

[25] 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.

[26] 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, 2002.

[27] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. 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.

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

[29] A. Ireland. A Practical Perspective on the Verifying Compiler Proposal. In *Proceedings of the Grand Challenges in Computing Research Conference*, 2004.

[30] G. T. Leavens, J. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In *Proc. of GPCE*, pages 221–235, 2006.