VICTORIA UNIVERSITY OF WELLINGTON
*Te Whare Wānanga o te Ūpoko o te Ika a Māui*

School of Engineering and Computer Science
*Te Kura Mātai Pūkaha, Pūrorohiko*

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

# Verifying Whiley Programs using an Off-the-Shelf SMT Solver

Henry J. Wylde

Supervisors: Dr. David J. Pearce, Dr. David Streader

October 22, 2014

Submitted in partial fulfilment of the requirements for
Engineering 489 - Engineering Project.

**Abstract**

This project investigated the integration of external theorem proving tools with *Whiley*—specifically, *Satisfiability Modulo Theories* (SMT) solvers—to increase the number of verifiable *Whiley programs*. The current verifier, the *Whiley Constraint Solver* (WyCS), is limited and hence there is a difficulty in verifying Whiley programs. This project designed and implemented an extension that supported the use of arbitrary SMT solvers with the *Whiley compiler*. The evaluation of this extension used the *Z3* SMT solver. The evaluation confirmed the value of using external SMT solvers with Whiley by emphasising the extension's ability to verify simple Whiley programs. However additional research would be required for applying this solution to more complex programs. This project also conducted an experiment that analysed WyCS's rewrite rules. This research may be used to educate WyCS's rewrite rule selection criteria, improving its verification abilities.

# Acknowledgments

I wish to thank my supervisors, Dr. David J. Pearce and Dr. David Streader, for all the time and effort they put into providing feedback, advice and help throughout this project. Their assistance has been invaluable.

I further wish to express my gratitude to those who helped proofread this report. Their feedback and comments helped to shape it and made a big difference.

# Contents

**1 Introduction** **1**
1.1 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**2 Background** **3**
2.1 The Whiley Project . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.1.1 The Whiley Programming Language . . . . . . . . . . . . . . . . . . 3
2.1.2 The Whiley Assertion Language . . . . . . . . . . . . . . . . . . . 4
2.1.3 The Whiley Constraint Solver . . . . . . . . . . . . . . . . . . . . 5
2.2 SMT Solvers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2.1 SMT-LIB . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2.2 Top SMT Solvers . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

**3 Design** **11**
3.1 The Whiley Project Architecture . . . . . . . . . . . . . . . . . . . . 11
3.2 Integration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2.1 The WyAL File Format . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2.2 The WyCS File Format . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2.3 Design Decision . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.3 Target Technology . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.3.1 SMT-LIB . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.3.2 Z3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.3.3 CVC4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.3.4 Design Decision . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

**4 Implementation** **17**
4.1 Encoding Native Data Types . . . . . . . . . . . . . . . . . . . . . . 17
4.1.1 Booleans, Unbounded Integers, Nulls and Unbounded Rationals . . . 17
4.1.2 Tuples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.1.3 Sets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
4.2 Encoding Native Functions . . . . . . . . . . . . . . . . . . . . . . . 21
4.2.1 Set Length . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
4.2.2 Subset and Proper Subset . . . . . . . . . . . . . . . . . . . . . . 23

**5 Evaluation** **25**
5.1 Experiment 1 – WyCS JUnit Tests . . . . . . . . . . . . . . . . . . . . 25
5.1.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
5.1.2 Limitations and Implications . . . . . . . . . . . . . . . . . . . . 26
5.1.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
5.1.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

5.2 Experiment 2 – Performance over WyCS JUnit Tests . . . . . . . . . . . . . . . . 27
5.2.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
5.2.2 Limitations and Implications . . . . . . . . . . . . . . . . . . . . . . 28
5.2.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
5.2.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.3 Experiment 3 – Whiley JUnit Tests . . . . . . . . . . . . . . . . . . . . . . . 29
5.3.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.3.2 Limitations and Implications . . . . . . . . . . . . . . . . . . . . . . 29
5.3.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.3.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.4 Experiment 4 – Whiley Benchmarks . . . . . . . . . . . . . . . . . . . . . . 32
5.4.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.4.2 Limitations and Implications . . . . . . . . . . . . . . . . . . . . . . 33
5.4.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
5.4.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
5.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

**6 Additional Experiment** **35**
6.1 WyCS Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
6.2 Experiment 5 – WyCS Rewrite Rules . . . . . . . . . . . . . . . . . . . . . . 36
6.2.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
6.2.2 Limitations and Implications . . . . . . . . . . . . . . . . . . . . . . 37
6.2.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
6.2.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

**7 Future Work and Conclusions** **43**
7.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
7.2 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

**Appendices** **47**

**A Evaluation Configuration Options** **47**

# List of Figures


1.1 GCD Function in Whiley . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

2.1 GCD Function Control Flow Graph . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 GCD Function Assertion in WyAL . . . . . . . . . . . . . . . . . . . . . . . . . 5

3.1 The Whiley Compiler Collection Architecture . . . . . . . . . . . . . . . . . . . . 13

5.1 Execution Time Comparison between WyCS and Z3 (Valid WyCS JUnit Tests) 30
5.2 Execution Time Comparison between WyCS and Z3 (Invalid WyCS JUnit Tests) 31

6.1 WyCS Rewrite Rule Usage (Passed WyCS JUnit Tests) . . . . . . . . . . . . . . 38
6.2 WyCS Rewrite Rule Usage (Failed WyCS JUnit Tests) . . . . . . . . . . . . . . . 39
6.3 WyCS Rewrite Rule Activation Probability (Passed JUnit Tests) . . . . . . . . . 40
6.4 WyCS Rewrite Rule Activation Probability (Failed WyCS JUnit Tests) . . . . . 41

# List of Tables


3.1 Comparison of Supported Data Types in Whiley, WyAL, WyCS and SMT-LIB 12

5.1 Test Coverage Comparison between WyCS and Z3 (WyCS JUnit Test Suite) . 27
5.2 Test Coverage Comparison between WyCS and Z3 (Whiley JUnit Test Suite) . 32
5.3 Performance Comparison between WyCS and Z3 (Whiley Benchmarks) . . . 34

# Chapter 1

# Introduction

*"I call [null references] my billion-dollar mistake"* —Tony Hoare [65]. *Null dereferencing* is the most common error in Java [19], but not the only error found within programming languages. Other common errors include *divide-by-zero*, *integer overflow* and *underflow*, and *invalid array indexing*. These errors may occur unexpectedly and are just some of the causes of failures in software applications. To understand the negative impact that software failures can have we only need to look at a few examples: the patriot missile friendly-fire incident, killing 28 soldiers and injuring many more people [16]; the NASA Mars surveyer program, where the Mars Climate Orbiter and the Mars Polar Lander were lost [72, 15]; and the Therac-25 medical electron accelerator that gave lethal radiation doses to patients [52]. The patriot missile incident was caused by a rounding error that worsened with time. The Mars Climate Orbiter was lost due to a mismatch between metric and imperial units of measurement, while the Mars Polar Lander was lost due to a sensor incorrectly indicating touch down when in fact the craft was still 40 meters away from the surface. The Therac-25 incident was caused by a counter overflow that prevented the software safety interlocks from activating.

Unit testing is a common method for discovering errors within software applications. Yet in 1972 Dijkstra claimed that *"program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence"* [24]. Mathematical and logical reasoning are methods for statically verifying code and were advocated by Hoare, Dijkstra and others [40, 25]. Such methods may be used by tools (e.g., FindBugs [4]) or verifying compilers (e.g., Dafny [68] and Chalice [50]) to certify the correctness of a program. A decade ago Hoare proposed a grand challenge for computer science as the development of a verifying compiler, where the compiler *"uses automated mathematical and logical reasoning methods to check the correctness of the programs that it compiles"* [41].

The Whiley programming language—developed by David J. Pearce at Victoria University of Wellington [62]—aims to realise the verifying compiler grand challenge. The language supports design-by-contract principles, where the programmer may write function contracts in the form of *pre-* and *post-conditions*. A pre-condition is a requirement that must hold on entry into the function, e.g., `b > 0` in Figure 1.1. A post-condition is a guarantee to callers about the result of the function on exit, e.g., `a % r == 0` and `b % r == 0`, in Figure 1.1. These contracts can be used in conjunction with Hoare logic [40] and theorem proving techniques to mathematically verify that the program is free of certain common errors. This facilitates the development of robust software applications [62, 63].

The Whiley Constraint Solver (WyCS) is the crux of the verification process in Whiley. It differs from the majority of theorem proving tools in its fundamental data types (discussed in § 3.2.2 and § 3.3.1), making it unique, yet not without limitations. For example, in order for WyCS to support *lists* and *maps*, complex encodings must be used which hinder its per-

```
function gcd(int a, int b) => (int r)
        requires a >= 0
        requires b > 0
        ensures r >= 0
        ensures a % r == 0
        ensures b % r == 0:

    if a % b == 0:
        return b

    return gcd(b, a % b)
```

**Figure 1.1:** An implementation of Euclid's greatest common divisor (GCD) algorithm [38] in Whiley. This function demonstrates the usage of pre-conditions (`requires`) and post-conditions (`ensures`).

formance and capabilities. An opportunity existed here to use an external verification tool for the verification of Whiley programs. Satisfiability Modulo Theories (SMT) solvers are descendent from SAT solvers and are one such class of verification tools [66]. Z3 is an example SMT solver, developed at Microsoft Research [21]. It is a high performance theorem prover that is maintained by a dedicated team of developers and won SMT-COMP [9] in both years it entered, 2007 and 2008 [10].

Using an external verification tool, such as the tried and tested Z3, could allow more Whiley programs to be verified. Thus, this project aimed to develop and evaluate a method for using such a tool in order to enhance the verification abilities of Whiley. Furthermore, this project evaluated and analysed some aspects of WyCS for future improvement of its verification abilities.

## 1.1 Contributions

This project has added and evaluated a verification extension to the Whiley project (1, 2) and provided research for future projects to enhance the verification abilities of WyCS (3). The main contributions are:

1. A method for using external theorem provers (i.e., SMT solvers) for the verification of contracts in Whiley programs (Chapters 3 and 4).

2. An evaluation of the above method using the existing Whiley test and benchmark suites (Chapter 5).

3. An analysis of the most frequently used and most likely activated rewrite rules in the Whiley Constraint Solver, allowing for informed future development of rewrite rule selection criteria (Chapter 6).

As of version 0.3.29, the verification extension was integrated with the Whiley project [57].

# Chapter 2

# Background

This chapter covers the necessary background information required to understand a) the Whiley project and b) Satisfiability Modulo Theories solvers, after which an overview of related work (programming languages and tools that provide code verification through theorem proving techniques) is given alongside how this project distinguishes itself.

## 2.1 The Whiley Project

The Whiley project was initiated in 2009 by David J. Pearce at Victoria University of Wellington [62]. In the past 5 years it has grown to be an excellent research tool in the area of code verification. Two of its key contributions are the Whiley programming language and the Whiley Constraint Solver, an automated theorem prover for verifying Whiley programs [60]. More information on the project can be found at its web page[1].

### 2.1.1 The Whiley Programming Language

Whiley is a hybrid imperative and functional programming language. It supports many of the modern language features that you might expect such as *unbounded integers*, *pure functions*, *sets* and *union types* [61]. Furthermore, Whiley incorporates a comprehensive *flow-sensitive* typing system that *"comes close to giving [Whiley] the flexibility of a dynamic language"* [58, 59]. Included in the flexibility of this typing system is the ability to automatically cast a variable to the appropriate type after a type check (with the `is` operator).

To demonstrate flow-sensitive typing, we provide the following code example that filters out integers from an arbitrary list. Specifically, it is demonstrated when `item` is automatically cast after the type check (line 4) from an `any` to an `int` within the *true* branch (line 5). Note that `[int]` is the type notation in Whiley for a list of `int`s.

```
1 function filterInts([any] items) => [int]:
2     [int] result = []
3     for item in items:
4         if item is int:
5             result = result + [item] // item has type int here
6
7     return result
```

One of the key features that distinguishes Whiley from many programming languages is verification. Whiley is designed to use mathematical and logical reasoning to certify the

[1] `http://whiley.org/`

correctness of programs. As introduced in Chapter 1, the programmer may write *pre-* and *post-conditions* on functions for verification. These conditions are translated into appropriate checks (assertions) in a format that resembles First Order Logic (FOL) (full elaboration on this may be found in § 2.1.2). Additionally, Whiley automatically generates assertions for common programming errors, such as invalid array indexing. For example, take the following Whiley expression: `int item = items[i]`. When compiling this expression, the following assertion (or similar) would be generated for verification:

```
assert "index out of bounds exception":
    forall ([int] items, int i):
        if:
            ...
        then:
            i >= 0
            i < |items|
```

The ellipsis under the `if` block denote the premises (line 4) and the statements under the `then` block are the checks (lines 6 and 7). The premises are omitted for brevity as they would depend upon the previous statements in and the pre-conditions of the function. This assertion checks that the variable `i` is within the bounds of the list `items`.

Unfortunately, given space constraints a more in-depth introduction to Whiley is out of the scope of this report. However the interested reader may consult the Whiley introductory tutorial [60] or language specification [61] for further information on Whiley.

### 2.1.2 The Whiley Assertion Language

The Whiley Assertion Language (WyAL) is a language for writing assertions of formulae. Each WyAL file contains a list of assertions that are designed to be verified by a verification tool.

The Whiley compiler collection contains a Verification Condition Generator which is responsible for reading a Whiley program and generating a list of assertions that—once verified—certify the correctness of the program. This generation process involves analysing the simple control flow paths of each function. To illustrate by example, Figure 2.1 shows the control flow graph for the GCD function defined in Figure 1.1 and highlights one of the simple control flow paths. The highlighted path represents the case when the conditional `a % b == 0` is `false`. The Verification Condition Generator analyses this path using Hoare logic [40] and generates an assertion that partially checks the function (i.e., the joint verification of all the generated assertions fully checks the function). Figure 2.2 illustrates the generated assertion; it uses the function pre-conditions (`a >= 0` and `b > 0`) and the logic inferred from the result of the conditional (`a % b != 0`) to verify one of the pre-conditions of calling `gcd(a, a % b)`, namely `a % b > 0`.

Individual assertions are also generated for cases other than just simple control flow paths and pre-conditions. For example, *array indexing* (to check the index's bounds), *division* (to check the denominator is not 0) and *return statements* (to check the function's post-conditions).

WyAL formulae are FOL-like, as can be seen inside the assertion of Figure 2.2. Furthermore, WyAL supports the same data types as Whiley, i.e., *lists*, *sets*, *maps*, *records*, *tuples*, etc.. One of the more interesting data types WyAL supports is *uninterpreted functions*. Uninterpreted functions are *pure*, that is, they have no side effects and calling them with the same arguments result in the same output. Additionally, they have no body—only parameter and return types. Their strength is in the ability to write assertions over them to enforce *what*

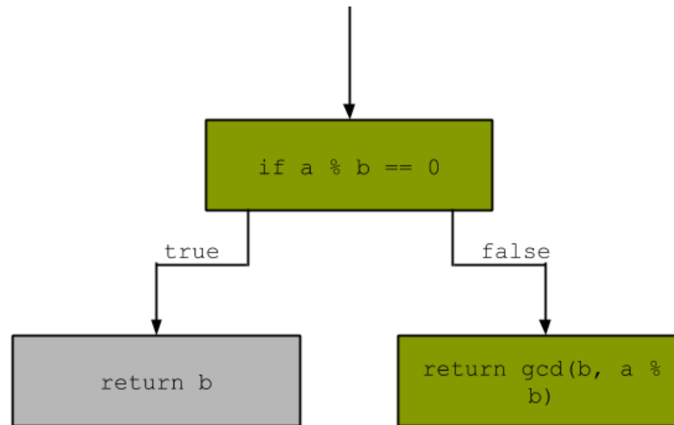


**Figure 2.1:** This figure shows the control flow graph for the GCD function in Figure 1.1. There are two simple control flow paths, one for each interpretation of the conditional (`true` and `false`). The highlighted path shows the sequence when the conditional `a % b == 0` is `false`.

```
1 assert "constraint not satisfied":
2     forall (int a, int b):
3         if:
4             a >= 0
5             b > 0
6             (a % b) != 0
7         then:
8             (a % b) > 0
```

**Figure 2.2:** A single assertion that partially verifies the GCD function in Figure 1.1. The assertion was generated from the simple control flow path when the conditional `a % b == 0` is `false` (Figure 2.1). The assertion is read as follows: for all integral values of `a` and `b`, if given the premises `a >= 0`, `b > 0` and `(a % b) != 0`, then check that `(a % b) > 0` holds.

behaviour (relationship between the arguments and output) is desired without specifying *how* that behaviour is achieved.

Take the following uninterpreted function representing the absolute function as an example: $abs(int\ a) \rightarrow (int\ r)$. At the moment if this function was used in an assertion then the output could be interpreted as any integer value, as no desired behaviour has been specified. To enforce a certain behaviour, an assertion over all the arguments is written for the function: $\forall\, a \in int \bullet abs(a) \geq 0$. This forces the output to be greater than or equal to 0, but the function behaviour is still weak as the function does not properly model an absolute function, e.g., the input argument of 3 could be related to the output of 5. To strengthen the behaviour, the following additional assertion is required: $\forall\, a \in int \bullet (abs(a) = a) \vee (abs(a) = -a)$.

### 2.1.3 The Whiley Constraint Solver

The Whiley Constraint Solver (WyCS) is a tool (SMT solver) for verifying a list of assertions, formally termed as *deciding their satisfiability*. A more in-depth discussion of how SMT solvers decide the satisfiability of formulae is coming up in § 2.2.

WyCS is able to read a WyAL file and verify its list of assertions. This process involves the following steps:

1. WyCS reads in a WyAL file and translates it into the WyCS file format.

2. Transformations are performed on the WyCS file (i.e., optimisations and simplifications such as *macro-expansion*).

3. *[Optional]* The WyCS file is written to disk.

4. The WyCS file is verified.

Step 1 mentions translating a WyAL file into the WyCS file format. This translation process requires elaboration as the WyCS file format only supports a subset of WyAL's language features. WyCS natively supports *booleans*, *unbounded integers*, *nulls*, *unbounded rationals*, *sets*, *tuples* and *uninterpreted functions*; it does not support the WyAL data types of *bytes*, *characters*, *lists*, *maps* and *records*. As such, the natively unsupported data types are encoded using WyCS's supported data types. This is explained further in § 3.2.2.

## 2.2 SMT Solvers

The Boolean Satisfiabilty Problem (SAT) is the problem of deciding the satisfiability of a propositional logic formula. The Satisfiability Modulo Theories (SMT) problem descends from SAT; it is the problem of deciding the satisfiability of formulae written using First Order Logic (FOL) theories. FOL extends propositional logic by including *predicates* and *quantifiers*. This area is vastly complex and a comprehensive introduction is out of the scope of this report, however the interested reader may wish to read [28] for a history of FOL and [29] for more on FOL and automated theorem proving.

A SMT solver is a verification tool that can decide the satisfiability of formulae written using FOL theories. A formula may either be *satisfiable*, where a model exists that satisfies them; or *unsatisfiable*, where no model exists that satisfies them.

To understand what this means, take the following formula (that uses a *number theory*): $x > 1 \land x < 5$. In this example, $\{x : 2\}$ is a model that proves the formula satisfiable. Now take $x > 5 \land x < 0$. It is not possible to fit any model to this formula, thus it is unsatisfiable.

SMT solvers are widely used in a range of applications including model checking and software verification [22]. In order to understand how SMT solvers may be used to verify software, we introduce the concept of formulae *validity* and *invalidity*. A formula is valid if all models satisfy it and invalid if there exists one model that violates it. Going back to the example satisfiable formula, $x > 1 \land x < 5$, it can be seen that this formula is also invalid as a model exists that violates it: $\{x : 0\}$. There is a relationship between these concepts: $valid \equiv \neg unsatisfiable$ and $invalid \equiv \neg satisfiable$. This relationship is important as in order to verify software, we desire the formula to be valid (all models satisfy it). Thus, using the aforementioned relationships, if the negated formula is unsatisfiable then the original formula is valid. A more detailed explanation of SMT solvers and their applications may be found at [66, 14].

### 2.2.1 SMT-LIB

SMT-LIB was initiated in 2003 and was at version 2 at the time of writing [11]. It contributes a standard language specification for SMT solvers and a set of benchmarks to enable the comparison of such solvers (both available at its web page[2]). The goal behind the standard language specification was to *"develop and promote common input and output languages for SMT solvers"* —Barrett *et al* [12]. As such, it defines a formal language that SMT solvers may support [11].

[2] `http://www.smt-lib.org/`

As the domain of SMT solvers uses different terminology to most progrmaming languages, we provide a brief overview of the important and commonly used terms throughout this report.

**Sorts** are closely related to types in traditional programming languages. Each term in SMT-LIB has a single sort. In contrast to types, there is no subtyping notion for sorts. It is possible to both *define* and *declare* new sorts. A sort definition creates an alias for another sort expression, e.g., `(define-sort AnotherNameForAnInt () Int)`. A sort declaration creates a new sort (also termed an uninterpreted sort), e.g., `(declare-sort Pair 2)`. The integer in `Pair` specifies how many type parameters the sort has. To declare the type of a term as a `Pair` two type arguments are used, i.e., `(Pair Int Int)`.

**Theories** are similar to abstract data types and their accompanying function definitions in programming languages. A theory defines a set of sorts and functions for use. For example, the *unbounded integer theory* defines an *addition* function. SMT-LIB defines a number of standard theories including but not limited to *booleans*, *unbounded integers*, *unbounded rationals* and *arrays*.

**A Logic** refers to a collection of theories for use. Not all theories are used by some formula, e.g., a formula may use booleans and integers but not rationals. It is thus possible to only use the theories that are actually required, potentially benefiting the performance of the decision process. An example logic is *QF_NIA* which refers to the theory of *quantifier-free integer arithmetic*.

The SMT-LIB theories of interest in this project were *booleans*, *unbounded integers*, *unbounded rationals*, *arrays*, *uninterpreted functions* (discussed in § 2.1.2) and *uninterpreted sorts*. Arrays in SMT-LIB are defined in the *arrays with extensionality theory*. Contrary to their name, they are more closely related to *maps* or *dictionaries* than *lists*. The array sort, `Array K V`, has two sort parameters: a key, `K`; and a value, `V`. It is possible to use any two sorts as the arguments to `Array`, enabling interesting interpretations as will be seen in § 4.1.3 for the encoding of sets. The arrays with extensionality theory also declares two functions, `store` and `select`. The `store` function takes an array, key and value as arguments and returns a new array. The new array mirrors the original with the exception that the specified key maps to the new value. The `select` function takes an array and key as arguments and returns the value that the key maps to.

### 2.2.2 Top SMT Solvers

The annual Satisfiability Modulo Theories Competition (SMT-COMP) was started in 2005 [10]. SMT solvers developed by numerous people and organisations compete to prove the satisfiability and unsatisfiability of a range of problems in different domains (logics), (e.g., quantifier-free problems vs. non-linear arithmetic problems). Here we mention some of the top SMT solvers from the first 6 years of competition results [10]. Note that some of the solvers overlap in the years they won because they won in different domains.

- **Z3** —winner 2007–2008. Z3 is a high performance theorem prover developed by Microsoft Research, available for both commercial and non-commercial use [21]. It first emerged in SMT-COMP in 2007 where it won 4 first and 7 second place awards. Since then, it has been at the forefront of top SMT solvers and been adopted into practice by languages and tools such as Spec# and Boogie [7, 23]. Given Z3's design for and

success in the verification of software it was a top contender as the external theorem proving tool for this project's extension.

- **CVC3** —winner 2006–2007 and 2009–2010. CVC3 is an open source theorem prover for SMT problems [13]. It has frequently been entered into SMT-COMP even with the development of its successor, CVC4 [8]. CVC4 bears resemblence to CVC3 only in namesake—it is a complete redevelopment of the tool with careful consideration with regards to architecture. CVC4 also showed promise in the years it entered into SMT-COMP [70]. From henceforth reference will be to CVC4 as it surpasses its predecessor in functionality and performance.

- **Yices2** —winner 2008–2009. Yices is a rich SMT solver developed by SRI International [26]. The current major version (2) built on version 1 significantly with important updates such as support for SMT-LIB (§ 2.2.1).

Our research into the competition winners provided a good summary of the top SMT solvers and will be referred to in § 3.3.

## 2.3 Related Work

There is slim tolerance for failure in software systems, especially safety-critical systems. When failure occurs, catastrophy often ensues [16, 72, 15, 52]. As such, the development of methods to ensure the correctness of software systems is an important research question [41].

Unit testing of software systems is great for discovering errors. However it is difficult and on its own, near impossible to guarantee the absence of bugs [24]. Formal model checking is an alternative that has been used to certify the correctness of flight-control systems [35], spaceflight-control systems [55], and more [17]. Event-B [1, 3]—evolved from the B method [69]—and the Rodin Platform [2] are examples of formal model checking tools. They have been used to formally verify safety-critical systems such as the Paris Métro [44, 43]. Formal modelling's strength is in verifying states of systems and their transitions, but not in checking for actual code errors. One area explored to aid this issue is code generation from a verified model [53, 27], but this is still an open research question and has limitations.

Use of theorem proving techniques is a method for code verification. Verifying compilers and verification tools that have explored this include Spec# [7], Dafny [45] and the Extended Static Checker for Java (ESC/Java2) [30]. These languages and tools all use SMT solvers to perform the checks. We now give an overview of each language or tool alongside some of their limitations and distinctions from Whiley.

### Spec#

Spec# is a programming language developed at Microsoft Research [7]. It extends the C# language with constructs to aid program verification. Some of the constructs it adds are *non-null types*, *pre-* and *post-conditions* and *object invariants*. The Spec# verifier (Boogie [5]) generates a list of assertions—requiring verification—in order to certify the correctness of the program.

In 2011 Barnett *et al* reflected on 6 years of experience with the Spec# project [6]. Many aspects were found to have worked quite well such as non-null types and Spec#'s ability to compile to a common platform. It was commented however that *"if one were to do the research project again, it is not clear that extending an existing language (here, C#) is the best strategy"*. This

was attributed to the language features that did not lend themselves to verification, e.g., *concurrency*. This is distinctly different to Whiley which was developed from scratch with verification in mind.

An additional key difference between Spec# and Whiley is modular arithmetic. Whiley supports *unbounded integers* and *unbounded rationals*, while Spec#—due to its C# base—has bounds on its *integers* and *floating points*. Bounded arithmetic can lead to issues such as floating point rounding errors [36]. Aside from the clear benefits of avoiding such unexpected behaviours, unbounded arithmetic also suits verification with its flexibility.

### Dafny

Dafny is a research language developed by Leino at Microsoft Research [45]. It follows an imperative paradigm with support for *objects* and other interesting data types such as *sets* and *sequences* [42]. The compiler contains a tool called the *Dafny verifier*. The Dafny verifier statically checks programs to certify their correctness [48, 39]. It is worth noting that as of 2010 Dafny uses Z3 [45], one of the top SMT solvers (as per SMT-COMP). With Z3, Dafny has made a significant contribution to the field of software verification tools [49].

Dafny's strength is in verifying the functional aspect of code. However it is limited in that it only verifies pseudo code: the compiler does not translate a Dafny program into executable code. Whiley differs in that it is aimed at being a complete compiler, as evident by its ability to compile to both Java and C.

Leino *et al* have contributed research on the use of theorem proving techniques for the verification of code [47, 46].

### ESC/Java2

ESC/Java2 is a verification tool developed at the Compaq Systems Research Center [30, 51]. This tool utilises theorem proving and program analysis techniques to find common programming errors (*null dereferencing*, *divide-by-zero*, etc.). In 2002 it used the SMT solver Simplify to perform these checks [30].

ESC/Java2 performs program analysis by reading in constraint annotations. For example, one such field constraint could be `/*@non_null*/ int[] data` which specifies that the field `data` may not be *null*. Further supported constraint annotations include *object invariants* and method *pre-* and *post-conditions*.

The authors feared that the tool *"could be too slow for interactive use"* —Flanagan *et al* [30]. As such, verification soundness was often sacrificed in the tool's development in order to improve the performance in areas such as *loop unrolling* and *object invariants*. With Whiley being designed for verification from the beginning, it removed some of the complications that caused this need for unsoundness. Performance however is still a concern as Whiley's verification tool (WyCS) can take some time to verify assertions.

# Chapter 3

# Design

This chapter covers the design decisions and the rationale for them within this project. This project involved developing an extension to enable the use of external theorem provers (SMT solvers) with Whiley. Thus, there were three key areas in particular that required careful consideration:

1. The Whiley project architecture and how the extension would integrate.
2. The input file format the extension would read for translation (e.g., WyAL or WyCS).
3. The output file format and target technology to use (e.g., SMT-LIB, Z3, CVC4, ...).

The extension required integration with the Whiley compiler collection (1) in order to translate an appropriate file format (2) into a format supported by an external SMT solver (3). The resultant translation would then be verified by the target external SMT solver. The translation stage posed a key challenge in this project due to the differences between the file format's supported data types, e.g., WyAL supports *lists* but WyCS and SMT-LIB do not. Table 3.1 details these differences and will be referred to in the upcoming sections. The table categorises each data type–file format pair as either a) natively supported, b) unsupported but may be *encoded* or c) unsupported. An encoding refers to representing one data type using one or more supported data types. For example, a *list* can be encoded as a set of *2-tuples* (i.e., pairs) where the first element in the tuple is the list element index. The use of encodings will be discussed further in the rest of the chapter.

## 3.1 The Whiley Project Architecture

Before beginning, some terminology is clarified here.

**The Whiley project** is a collection of tools—or plugins—that may be used to compile, verify and run Whiley applications.

**The Whiley compiler collection** is the framework that combines the plugins to create a harmonious compiler.

**The Whiley compiler** is the plugin used in the Whiley compiler collection to compile Whiley files into Whiley Intemediate Language (WyIL) files.

**A Whiley program** is a piece of software written in the Whiley language.

| Data Type | Whiley | WyAL | WyCS | SMT-LIB |
|---|---|---|---|---|
| **Boolean** | ✓ | ✓ | ✓ | ✓ |
| **Byte** | ✓ | ✓ | ✗ | ✗ |
| **Character** | ✓ | ✓ | ✗ | ✗ |
| **Integer** | ✓ | ✓ | ✓ | ✓ |
| **Null** | ✓ | ✓ | ✓ | ✗ |
| **Rational** | ✓ | ✓ | ✓ | ✓ |
| **List** | ✓ | ✓ | ✗ | ✗ |
| **Map** | ✓ | ✓ | ✗ | ✓ |
| **Record** | ✓ | ✓ | ✗ | ✗ |
| **Set** | ✓ | ✓ | ✓ | ✗ |
| **Tuple** | ✓ | ✓ | ✓ | ✗ |
| **Any** | ✓ | ✓ | ✓ | ✗ |
| **Void** | ✓ | ✓ | ✓ | ✗ |
| **Negation** | ✓ | ✓ | ✓ | ✗ |
| **Intersection** | ✓ | ✗ | ✗ | ✗ |
| **Union** | ✓ | ✓ | ✓ | ✗ |
| **Function** | ✓ | ✗ | ✗ | ✗ |
| **Recursive** | ✓ | ✗ | ✗ | ✓ |

**Table 3.1:** Comparison of supported data types in Whiley, WyAL, WyCS and SMT-LIB. A *white* entry marks a natively supported data type, a *cyan* entry marks an unsupported data type but may be supported with the use of an encoding, and a *red* entry marks an unsupported data type.

The Whiley project is designed as a modular collection of plugins. Each plugin has a different purpose, e.g., translating one file format into another or validating a file. A list of the main plugins and their purposes follow:

- **Whiley Compiler (WyC)** —responsible for syntactically and semantically validating a Whiley file before translating it to a WyIL file. This plugin also performs all the necessary pre-processing steps—such as *name resolution*—on the Whiley file. The WyIL file is a much more managable intermediate representation for Whiley programs that supports pipeline application. Some pipelines that are applied are *constant propagation*, *dead code elimination* and *live variable analysis*. Subsequent plugins then utilise the WyIL file as their input.

- **Whiley Java Compiler (WyJC)** —responsible for translating a WyIL file into a Class file [64]. The Class file may then be executed on the Java Virtual Machine with the assistance of a provided runtime library.

- **Whiley C Compiler (WyCC)** —responsible for translating a WyIL file into a C file. The C file may then be natively compiled and executed using a tool such as the GNU Compiler Collection [71].

- **Verification Condition Generator** —responsible for reading a WyIL file and generating a list of assertions that will—once verified—certify the correctness of the original Whiley file. The list of assertions are written out into a WyAL file for successive plugins to read and verify. Section 2.1.2 introduced and explained the verification condition generation process.

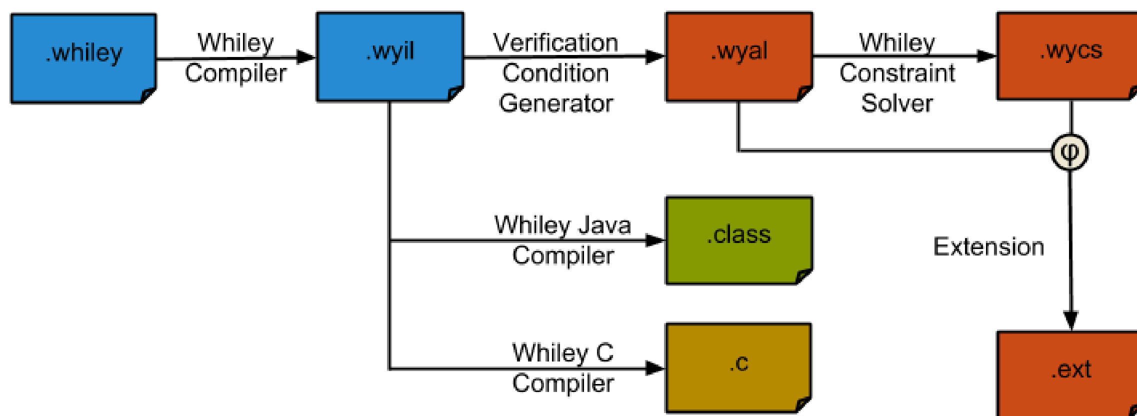


**Figure 3.1:** The Whiley compiler collection architecture. This figure shows the relationship between the plugins and their input/output file formats. The *blue* file formats are critical to most of the plugins and the *orange* file formats are related to verification. At the time of design, a choice needed to be made about which file format this project's extension should integrate with: either WyAL or WyCS. This opportunity is illustrated by the $\varphi$ decision symbol and is discussed in § 3.2.

- **Whiley Constraint Solver (WyCS)** —responsible for certifying the correctness of a Whiley program by verifying its list of assertions. This verification process is led by the translation of the WyAL file into a WyCS file. This is required as WyCS only supports a subset of WyAL's language features (discussed in § 3.2.1 and § 3.2.2).

The relationship between these plugins is shown in Figure 3.1, along with an extra node representing this project's extension. The plugin nature of the Whiley project made the development of an extension elementary. The structure of the Whiley compiler collection made the integration of such an extension just as straightforward. Thus, the extension can be viewed as the following plugin:

- **Extension** —responsible for certifying the correctness of a Whiley program by verifying its list of assertions. This plugin provides an alternative to WyCS for the certification of a Whiley program.

The node in Figure 3.1 demonstrates this new plugin and illustrates a key design decision regarding the integration of the extension (discussed in § 3.2).

## 3.2 Integration

The Verification Condition Generator reads in a WyIL file and generates a list of assertions for verification from the Whiley function contracts. This list is available for input in either the WyAL or the WyCS file format. Hence, this marked the two possible integration points for the extension:

- After the WyAL file format is generated.

- After the WyCS file format is generated.

What follows is an analysis of the differences in these file formats and the pros and cons of integrating at each point.

### 3.2.1 The WyAL File Format

The WyAL file format aims to be FOL-like while retaining the majority of data type information from the original Whiley source. It can be seen in Table 3.1 that it supports almost all of Whiley's data types, with the exclusion of *intersections*, *functions* and *recursive types*.

The main pros and cons of using the WyAL file for the integration point would be that (almost) no information has yet been lost. It would be up to the translator and target technology to choose how to represent each of the data types. For example, *maps* and *lists* could have a one-to-one mapping to the native array data type (§ 2.2.1).

During translation to the WyCS file format, the WyCS plugin also performs optimisations and simplifications to the code. Thus, one disadvantage of integrating at the WyAL level would be that some optimisations would not have yet been performed, e.g., *macro expansion*. In addition, any optimisations added to WyCS in the future would also be excluded, meaning that if they were desired, modification of the extension would be required.

### 3.2.2 The WyCS File Format

The WyCS file format has had optimisations and simplifications performed prior to translation. It natively supports *sets* and *tuples* as its base complex data types. Thus, some data types available at the WyAL level are encoded using these when translating from WyAL to WyCS. Table 3.1 shows all of the data types that require some form of encoding. These data types and encodings are as follows:

- **Bytes** are encoded as *integers*.
- **Characters** are encoded as *integers*.
- **Lists** are encoded as sets of *2-tuples*, where the first element in the tuple is the list element index. This encoding includes conjectures that enforce common properties of a list, e.g., *continuity*: a list containing an element at index `n` also contains elements at the indicies `0` through to `n - 1`.
- **Maps** are encoded as sets of *2-tuples*, where the first element in the tuple is the map key. This encoding includes conjectures that enforce common properties of a map, e.g., *uniqueness*: a map key only occurs once.
- **Records** are encoded as *n-tuples*, where each tuple element corresponds to a field in the record. This encoding is straightforward as both records and tuples are finite sets of (differently) typed elements.

Using the WyCS file format as the input for the translation would alleviate the need to encode all data types. *Lists*, *maps* and *records* would all be pre-encoded and come with property conjectures. Furthermore, any changes that are made at the WyCS level would immediately be reflected in the external SMT solver without direct alteration of the extension.

The drawback is no flexibility in their encoding. *Maps* and *lists* would already be encoded as sets of tuples, which means that after encoding sets and tuples in the target language (e.g., SMT-LIB), maps and lists would effectively be encoded twice. This would be inefficient and would introduce performance limitations. For example, maps require a conjecture enforcing unique keys, whereas if they were directly encoded as arrays they would not require this (§ 2.2.1).

### 3.2.3 Design Decision

The differences between the two decisions was clear. Integrating at the WyAL file format would provide flexibility in how the data types are encoded, while at the WyCS file format, the extension would utilise pre-existing code and prior well-reasoned encoding decisions; the negative consequence being that some data types may effectively be encoded twice.

The concluded decision was to integrate at the WyCS file format. The key reason behind this was that it utilised pre-existing optimisations and simplifications. We believed that the double encoding issue would not have a huge impact on performance as sets seemed a natural base data type and had efficient encodings as arrays (discussed in § 4.1.3).

## 3.3 Target Technology

In order to improve the verification abilities of Whiley we wanted to use a top of the line external theorem proving tool. Our hyphothesis was that it would be more efficient and bug-free than the current verification tool, WyCS.

Section 2.2.1 introduced SMT-LIB, a standard language specification for SMT solvers. SMT-LIB would provide a common interface for integrating the extension with external SMT solvers. As such, the inclusion of a language specification as one of the possible target technologies meant the extension would be portable.

SMT-COMP provided a comprehensive listing of the current top performing SMT solvers. It further summarised the SMT solvers by their aptitude in different theories and logics. To recap from § 2.2.1, a logic refers to a group of theories, e.g., *unbounded integers* and *arrays*. Thus, it was possible to narrow down the list of viable SMT solvers by analysing the requirements of the WyCS data types. This narrowed list included Z3, CVC4 and Simplify—all of which support SMT-LIB.

In addition, the research of Barnett *et al* with verifying compilers provided findings of prior experience on using SMT solvers in the verification of software: specifically the success of Z3 with Dafny and Spec# [45, 6] and the use of Simplify with ESC/Java2 [51]. Prior to 2007 Dafny and Spec# also used Simplify. However De Moura and Bjørner note a substantial improvement in performance after moving to Z3 [21].

Thus, three main options presented themselves as candidates for the extension's target technology. These options were: the language specification SMT-LIB and the SMT solvers Z3 and CVC4. A discussion of their pros and cons follow.

### 3.3.1 SMT-LIB

SMT-LIB provides a language specification for SMT solvers (§ 2.2.1). Both Z3 and CVC4 support SMT-LIB [20].

Using a standardised language specification would mean that the end implementation was portable between different SMT solvers—provided they support it. It would add a level of indirection, decoupling it from any one external SMT solver. Thus, it would be easy to swap out one solver for another, or even to use multiple solvers to verify a Whiley program. In such cases, the Whiley compiler collection could attempt to use one solver, but if it *timed-out* or was unable to solve the list of assertions, it could try use another. While this concept was out of the scope of this project, it was interesting to keep in mind as a possibility for future work.

SMT-LIB supports the common language features of *booleans*, *unbounded integers*, *unbounded rationals*, *arrays*, *uninterpreted functions* and *uninterpreted sorts*. This language feature

set highlights one negative consequence of using SMT-LIB; other SMT solvers support a superset of SMT-LIB, e.g., Z3 and CVC4 support records. It should be noted however that SMT-LIB's language features are comprehensive and sufficient to encode most constraint problems. Language features such as records can be encoded in SMT-LIB with low impact on performance.

### 3.3.2 Z3

Z3 is a high performance theorem prover developed by Microsoft Research [21] (discussed in § 2.2.2). It is targeted at verifying assertions that are generated from program constraints, thus making it an ideal target technology for this project.

Z3 supports a superset of the SMT-LIB language features. Some of the more interesting data types it supports above and beyond SMT-LIB are *records*, *enumerations* and *recursive types*. Recursive types are of particular interest as they are present in Whiley but not (yet) WyAL and WyCS. Particular details of how Z3 handles recrusive data types are not covered in this report, but their existence is important for future work (§ 7.1).

### 3.3.3 CVC4

CVC4 is an open source theorem prover developed in collaboration by New York University and University of Iowa [8] (discussed in § 2.2.2). It is the successor of CVC3 and due to its total redevelopment, has a highly optimised internal design.

Similar to Z3, CVC4 is also a superset of SMT-LIB and further supports *records* and *recursive types*. Interestingly, CVC4 additionally supports *function types*. Function types are only used at the Whiley level, so while they are not currently required, they would be available if WyAL and WyCS ever wish to support them. One key language feature that CVC4 does not support is *function overloading*. While function overloading is not necessary (e.g., the function's parameter types could be mangled into its name), it definitely would help make the translation simpler and clearer.

### 3.3.4 Design Decision

SMT-LIB was chosen as the target technology. A standard language specification was ideal for this project; it helped future proof the extension by decoupling it from any one external SMT solver. Thus, if a new SMT solver showed promise or an old one was discontinued, it would be simple to change the target SMT solver.

SMT-LIB was the target technology, but an external SMT solver was still required for the verification of the extension's output. Z3 was chosen as this solver for the development and evaluation of the extension. The two main reasons for this were:

1. Z3's success with other verifying compilers (Dafny [45], Spec# [6]).

2. Z3 supports function overloading while CVC4 does not.

This decision should have slim impact on the implementation of the extension. The extension was designed to be decoupled from any one external SMT solver such that users of the extension are not limited to Z3 and may equally use CVC4 themselves.

# Chapter 4

# Implementation

This chapter discusses key implementation details of the extension. To recap from Chapter 3, *"the extension required integration with the Whiley compiler collection in order to translate an appropriate file format into a format supported by an external SMT solver"*. The integration with the Whiley compiler collection was straightforward due to its modular design, thus this chapter focuses on the translation stage, specifically, translating files written in the WyCS file format into the SMT-LIB format. This may be broken up into two categories:

- Encoding the WyCS native data types that are unavailable in the SMT-LIB file format.
- Encoding the WyCS native functions and operations that are unavailable in the SMT-LIB file format.

## 4.1 Encoding Native Data Types

There is a mismatch between the natively supported data types of WyCS and SMT-LIB. Specifically, WyCS supports *booleans*, *unbounded integers*, *nulls*, *unbounded rationals*, *sets*, *tuples* and *uninterpreted functions*. SMT-LIB supports *booleans*, *unbounded integers*, *unbounded rationals*, *arrays*, *uninterpreted functions* and *uninterpreted sorts*.

WyCS further supports the *any* data type. Although it is not possible to declare a value as *any*, it is important in subtyping relations. However, the use of *any* and subtyping was considered out of scope of this project and is discussed in future work (§ 7.1).

The following sections thus discuss the translation and encoding of WyCS's native data types (excluding `any`) into SMT-LIB.

### 4.1.1 Booleans, Unbounded Integers, Nulls and Unbounded Rationals

The WyCS primitive data types (booleans, unbounded integers and unbounded rationals) all had direct mappings to equivalent sorts in SMT-LIB. The respective equivalent sorts were `Bool`, `Int` and `Real`. Nulls were not natively supported, but were represented as a singleton of a newly declared sort:

```
1 (declare-sort Null 0)
2 (declare-fun NULL () Null)
```

Basic operations such as *boolean conjunction* and *integer addition* are also natively supported in SMT-LIB. However one unsupported operation in SMT-LIB is *integer remainder*, despite the fact that the majority of SMT solvers support it. This presented a difficulty in the implementation of the extension. Ideally the extension was to be decoupled from any one

external SMT solver yet, in order to support integer remainder, a dependency on individual SMT solvers' syntaxes would be required. Unfortunately this was unavoidable. Despite this, minimizing the coupling of the extension to individual SMT solvers still remained a priority.

### 4.1.2 Tuples

Tuples are an immutable, ordered list of elements. The elements may have the same or different types, e.g., `r = (5, true)` is a *2-tuple* of an *integer* and *boolean*.

WyCS has native support for tuples; they are a core data type. SMT-LIB does not natively support them and hence, they required an encoding. It is worth mentioning that some SMT solvers provide native support for tuples (Z3 [54] and CVC4 [56]—to name two), however as the target technology was the SMT-LIB format, these native features were not used.

Two options for encoding tuples were explored: using *name mangling* and using an uninterpreted sort. What follows is a discussion of these two encodings with the final implementation decision and reasons.

#### Encoded with Name Mangling

The immutable nature of tuples opened up the possibility of encoding them into SMT-LIB with name mangling. In this method, a $n$-tuple would have $n$ variables declared, one for each of its elements. For example, taking the above 2-tuple, `r`, the two elements (at indices 0 and 1) would be declared as two distinct variables:

```
1 (declare-fun r_0 () Int)
2 (declare-fun r_1 () Bool)
```

Note that a variable in SMT-LIB is just an uninterpreted function without any parameters. The main benefit of this method would be the lack of additional uninterpreted sorts (§ 2.2.1). Uninterpreted functions and sorts are frequently accompanied by quantifiers: quantifier instantiation is a well-known issue for SMT solvers [67, 33, 32].

A limitation with name mangling is the inability to pass a tuple as a single value, e.g., for a single assignment to another tuple or for an addition to a set. This would not affect assignment or equality statements as they may be written in full using a conjunction, but it would affect the use of tuples as sort arguments—as is needed in the case of sets. To illustrate this, take the following variable declaration where `rs` is declared as a set of `?`s:

```
1 (declare-fun rs () Set ?)
```

The set sort takes a single sort argument (§ 2.2.1) that specifies its element's sort, yet there is no valid possible tuple sort to use here. Hence, with name mangling it is not possible to declare a set of tuples. This was considered a key limitation in this encoding method.

#### Encoded as an Uninterpreted Sort

Tuples may also be encoded as uninterpreted sorts (§ 2.2.1), wherein a $n$-tuple would be encoded as a sort with $n$ sort parameters. The sort parameters would indicate the sorts of the tuple's elements. We illustrate here an example declaration of a 2-tuple sort in SMT-LIB.

```
1 (declare-sort Tuple2 2)
```

This sort requires two sort arguments whenever used. For example, here is a declaration of a variable as a 2-tuple:

```
1 (declare-fun r () (Tuple2 Int Bool))
```

Here the variable `r` has been declared as a 2-tuple of an integer and boolean (referred to in this report as an *interpretation* of the sort). The sort declaration may be used for other interpretations of 2-tuples, e.g., a 2-tuple of a boolean and boolean. Thus the initial 2-tuple sort declaration does not need repeating. However, other tuple sizes would still require new sort declarations (e.g., a 3-tuple).

Nothing is yet known about this new sort; in order to work with the tuple, corresponding element retrieval functions would need declaration. This involves declaring a retrieval function for each tuple element in a given tuple sort interpretation. Here we give some example retrieval function declarations and usages:

```
1 (declare-fun get0 ((Tuple2 Int Bool)) Int)
2 (declare-fun get1 ((Tuple2 Int Bool)) Bool)
3
4 (assert (= (get0 r) 0))
5 (assert (= (get1 r) true))
6 (assert (not (= (get0 r) (get1 r))))
```

The integer suffix on the functions' names represents the index of the element for retrieval. Uninterpreted functions in SMT-LIB are pure, that is, they have no side effects and are predictable (§ 2.1.2). As such, this enforced the immutability property of tuples; once an element inside the tuple is set it cannot be changed.

Notice how the `get` functions take the tuple sort interpretation (`Tuple2 Int Bool`) as opposed to a generic type (e.g., `Tuple2 A B`). This is because while there is support for sort parameters in uninterpreted sorts, there is no support for generics in uninterpreted functions. Thus, these retrieval declarations would require repetition for other interpretations of the tuple sort.

A consequence of using uninterpreted sorts for the encoding is that there is no information known about them. This means there is no notion of equality between two tuples and one must be asserted:

```
1 (assert (forall ((tuple0 (Tuple2 Int Bool)) (tuple1 (Tuple2 Int Bool)))
2                 (xor (and (= (get0 tuple0) (get0 tuple1))
3                           (= (get1 tuple0) (get1 tuple1)))
4                      (distinct tuple0 tuple1))))
```

As alluded to in the previous section, this use of an uninterpreted sort requires a quantified conjecture (to enforce the notion of equality over all tuples). Furthermore, the number of quantified conjectures only increases with the addition of more tuple sort interpretations. Given the difficulty of quantifier instantiation for SMT solvers [67, 33, 32], this becomes the main limitation in this encoding method. The consequence of this is the occasional inability of SMT solvers to decide the satisfiability of a list of assertions.

One benefit of this method however would be the ability to use tuples as single values, e.g., having sets of tuples.

#### Implementation Decision

The implementation chosen was to use uninterpreted sorts to encode tuples. The reason for this was the fact that this option supported sets of tuples, while name mangling did not. Name mangling may have had efficiency benefits owing to the lack of quantifiers, but the comprehensive usage of tuples as set elements meant that name mangling had to be discarded as a viable option.

### 4.1.3 Sets

Sets are unordered collections of elements that do not contain duplicate elements (i.e., each element is unique). For example, `r = {0, 1}` is a set with 2 elements and is semantically the same as `r = {1, 0, 1}`.

WyCS natively supports sets as one of its core data types. They are used in conjunction with tuples to represent other complex data types, such as *lists* and *maps*. In contrast, SMT-LIB does not natively support sets, rather it supports arrays (§ 2.2.1).

Two possible encodings for sets were explored in this project and are explained below before the implementation decision and reasons are summarised.

#### Encoded as a Cons Operator

A set may be encoded as the concatenation of a *head* element and *tail* set. This is generally referred to as the *cons(truct)* operator and is described in detail by Bradley *et al* [18]. It is more commonly used for lists, but could equally be applied to sets, although extra care would need to be taken when adding and removing elements to sets in order to maintain their uniqueness property. The following demonstrates how the set, `r`, would look in SMT-LIB:

```
1 (assert (= r (cons 0 (cons 1 empty))))
```

As shown, this encoding provides an easy mechanism for building up set constants, yet it faces similar limitations to the encoding of tuples as uninterpreted sorts (§ 2.2.1). Quantified conjectures would be required to include the notion of equality. Furthermore, complications could arise when adding and removing elements from sets due to their uniqueness property. Either a check for containership would need to be done before the addition of elements or the removal function would need to recurse through the set to remove all instances.

#### Encoded as Arrays

Sets could also be encoded as arrays in SMT-LIB. In this method a set would be encoded as a defined sort—an array of `T` to `Bool`[1]:

```
1 (define-sort Set (T) (Array T Bool))
```

This defines an alias for an array that has 1 sort parameter, `T`. Here, the boolean value would be used to represent whether an element is contained within the set (`true`) or not (`false`). Any sort argument may be provided when interpreting the set sort definition. For instance, we could declare `r` as a set of integers:

```
1 (declare-fun r () (Set Int))
2 (assert (= r (add (add empty 0) 1)))
```

The variable `r` is built up using the `add` function and the `empty` constant. The `add` function and `empty` constant would require definitions for each interpretation of the set sort.

```
1 (define-fun add ((set (Set Int)) (t Int)) (Set Int)
2                 (store set t true))
```

The `add` function utilises the pre-existing function for arrays, `store` (§ 2.2.1). In this example, the `store` function returns a new array with the key, `t`, mapped to the new value, `true`. This means the `add` function would also return the resultant set after the operation. For brevity, two additional functions `remove` and `contains` have been omitted, but

[1] Z3 represents sets similarly in their tutorial [54]

`remove` would be defined similar to `add` and `contains` would use the second pre-existing function for arrays, `select` (§ 2.2.1).

The uniqueness property of sets would be enforced when the set element is used as the key in the array—a key is mapped to exactly one value at a time. Thus, `r` could also have been built up as:

```
1 (assert (= r (add (add (add empty 1) 0) 1)))
```

Furthermore, arrays have a pre-defined notion of equality. This would be inherently utilised in this encoding, providing a potential performance benefit over the cons encoding method.

In the prior examples a set was built up using the `empty` constant as the base. This constant would be declared as follows:

```
1 (declare-fun empty () (Set Int))
2 (assert (not (exists ((t Int))
3                         (contains (as empty (Set Int)) t))))
```

The declaration includes a quantified conjecture to say that the `empty` constant is in fact, empty. Overloading the `empty` constant's name caused some issues in expressions where its type could not be inferred. For example, when building up `r`, it may be difficult to infer the type of `empty` as `Set Int`. WyCS had already determined the type of these expressions and thus, these could be used to explicitly qualify `empty`'s type, e.g., `(as empty (Set Int))`. A limitation would still exist for examples where the type of `empty` was not previously determined, e.g., `(length empty)`.

#### Implementation Decision

The implementation chosen was to encode sets as arrays in SMT-LIB. The main reason for this was that it allowed us to capitalise on pre-existing functions such as `store` and `select`. Furthermore, the uniqueness property and equality notion would be naturally enforced, removing the need to include additional quantified conjectures.

## 4.2 Encoding Native Functions

WyCS supports three native functions that do not have equivalents in SMT-LIB. These are the set *length*, *subset* and *proper subset* functions. Thus, each was encoded as either a defined or uninterpreted function. These function encodings are described in detail in the upcoming sections.

### 4.2.1 Set Length

WyCS sets were encoded in SMT-LIB using arrays (§ 4.1.3). Arrays do not provide any notion of length, yet set length is a native function of WyCS. Thus, one must be defined (for each interpretation of a set sort) as an uninterpreted function:

```
1 (declare-fun length ((Set Int)) Int)
```

This function was conceptually defined to take a set and return its length. The length of a set can be thought of as having the following properties:

1. A set's length is 0 or more.

2. A set's length is proportional to the number of elements within it.

Property 1 was easy to assert.

```
1 (assert (forall ((set (Set Int))) (<= 0 (length set))))
```

Property 2 was more difficult. In SMT-LIB, there is no method for retrieving the number of elements in an array. Thus Property 2 had to be defined inductively.

```
1 (assert (forall ((set (Set Int)))
2         (=> (distinct set (as empty (Set Int))) ; (a)
3             (exists ((t Int))                   ; (b)
4                    (and (contains set t)
5                        (= (length set)
6                           (+ 1 (length (remove set t)))))))))
```

This conjecture states that a set is either empty (a) or there exists some element, `t`, contained within it (b). In the latter case (b), the sets length is hence equal to `1` plus the length of the set excluding `t`.

These two conjectures contain some quantifiers: a `forall` in each and a nested `exists` in Property 2's conjecture. This introduces two limitations when verifying any constraints that utilise this set length encoding. The first pertains to performance and quantifier instantiation (as explained under *Encoded as an Uninterpreted Sort* in § 4.1.2). The second pertians to the ability to check for satisfiability. SMT solvers will now sometimes produce the result *unknown* when checking for the satisfiability of the constraint; they can however still check—comparatively easily—for the unsatisfiability.

Take the following example set from the previous section:

```
1 (declare-fun r () (Set Int))
2 (assert (= r (add (add empty 0) 1)))
```

If we desired to check the length of `r`, we would have to check for unsatisfiability as follows:

```
1 (assert (distinct (length r) 2))
2 (check-sat)
```

Note that the `(check-sat)` directive gets the SMT solver to decide the satisfiability. This means it may return *unsatisfiable* or *unknown*, as well as *satisfiable*.

If we had attempted to check for satisfiability (i.e., `(assert (= (length r) 2))`), then the SMT solver would have returned unknown. However as we were checking for unsatisfiability, the SMT solver would return unsatisfiable.

The first limitation (a decrease in performance due additional quantifiers) hinders the verification ability of the extension. However the second limitation (may only check for unsatisfiability) does not affect the verification of Whiley programs. This is because during verification we require formula validity and as explained in § 2.2. Thus, we only need to check for unsatisfiability and not satisfiability.

This length function encoding works well for simple problems, yet it is still subject to the weaknesses of FOL. In particular, it is difficult to have a finite set of axioms that state all properties of sets and their lengths. As an example, there is a relationship between sets' lengths and the operations performed on them (e.g., *union* and *intersection*). This is referred to as the *inclusion-exclusion principle*: $|A \cup B| = |A| + |B| - |A \cap B|$. Suter *et al* have developed a decision procedure for sets with length constraints in SMT problems [73]. However their research had boundaries that sets are not constrained by in the Whiley language. For example, their research was constrained to sets of integral values.

### 4.2.2 Subset and Proper Subset

WyCS has native support for the subset and proper subset functions. These functions take two sets and check to see whether they are (proper) subsets (returning *true* if they are and *false* otherwise). The definition for the proper subset function is as follows:

```
1 (define-fun subseteq ((first (Set Int)) (second (Set Int))) Bool
2         (forall ((t Int))
3                 (=> (contains first t) (contains second t))))
```

The proper subset function (subseteq) states that if first is a proper subset of second, then every element in first is also contained within second.

The subset function can be defined as a proper subset with an extra condition, that the two sets are distinct.

```
1 (define-fun subset ((first (Set Int)) (second (Set Int))) Bool
2         (and (subseteq first second)
3              (distinct first second)))
```

An alternative way of saying two sets are distinct would be to check their lengths; if their lengths are different then clearly the sets must be different. This alternative was briefly considered before we recognised that it would utilise the length function, which contains nested quantifiers. The equality notion for sets (arrays) on the other hand would only use a top-level quantifier. Thus we believed the equality definition would be more efficient.

Similar to the union and intersection functions, there are relationships between the subset and proper subset functions and sets' lengths. Specifically, if a and b are sets and a is a subset of b, then the length of a must be less than or equal to the length of b. A similar relationship exists for the proper subset of two sets. Mathematically these relationships are written as follows: $a \subset b \implies |a| < |b|$ and $a \subseteq b \implies |a| \leq |b|$. In order to enforce this relationship extra conjectures would be required. While such conjectures were written for the extension, we found that they introduced too many quantifiers and verification either returned unknown or timed-out. As such, they were excluded from the final version of the extension.

# Chapter 5

# Evaluation

This chapter details the evaluation method for the verification extension of the Whiley project. The main aim of the extension was to enhance the verification abilities of Whiley. Four experiments were run in order to evaluate the solution developed. The experiments utilised pre-existing JUnit test suites and benchmarks from the Whiley project.

The first and second experiments evaluated and compared the extension's accuracy and execution time with WyCS's on the WyCS JUnit test suite[1]. The WyCS test suite is written in the WyAL file format and is aimed at explicitly testing the verification abilities of WyCS. Results from these experiments reflected the correctness of the extension's translation process and the efficiency of the translated formulae.

The third experiment looked at a more complete picture with the Whiley JUnit test suite[2]. The Whiley test suite is written in Whiley and tests the compiler collection as a whole (compilation and verification). Results from this experiment reflected the verification abilities of the external SMT solver on Whiley programs, i.e., how well the external SMT solver handled more detailed and representative translations.

The fourth experiment evaluated and compared the extension with WyCS on the Whiley benchmarks[3]. The Whiley benchmarks are also written in Whiley and test the compiler collection as a whole. Results here further reflected the verification abilities of the external SMT solver on Whiley programs.

## 5.1 Experiment 1 – WyCS JUnit Tests

The WyCS JUnit tests are written in the WyAL file format. They explicitly test native language features of WyCS such as integer operations (addition, subtraction, etc.) and set operations (subset, length, etc.). Thus, this experiment directly tested the completeness of the extension as well as its ability to correctly translate WyCS's language features.

### 5.1.1 Methodology

The test suite contains a *valid* and an *invalid* test group. Tests in the valid group were expected to be compilable and verifiable. Tests in the invalid group were expected to be either uncompilable or unverifiable; they were used to check that an error is reported when the formulae in the tests are known to be wrong.

[1] `https://github.com/Whiley/WhileyCompiler/tree/master/modules/wycs/tests/`
[2] `https://github.com/Whiley/WhileyCompiler/tree/master/tests/`
[3] `https://github.com/Whiley/WyBench/`

The experiment aimed to evaluate the extension's translation accuracy and performance against WyCS. Z3 was used with the extension for the reasons explained in § 3.3. At the time of evaluation, Z3 was at version 4.1 and Whiley and WyCS were at version 0.3.27.

All of the tests within each test group were selected for the experiment unless they failed due to issues with the Whiley compiler (as opposed to the verification tools). These tests were removed prior to the experiment to allow for a fair evaluation of just the verification tools. Furthermore, some of the tests caused WyCS to get *stuck*, i.e., enter an infinite loop. These tests were ignored for just WyCS and were treated as failed tests. The selected tests for each test group were run with each verification tool (WyCS and Z3) and the results recorded.

A variety of configuration options are available for WyCS and the extension. This experiment used the default compiler configuration options and are included as an appendix (Appendix A). A full explanation of the options and their meaning is considered out of the scope of this project, however a partial understanding of WyCS's options may be gained by reading § 6.1.

The verification tools ideally terminate with the result of either satisfiable or unsatisfiable. In the event that a tool is unable to decide the satisfiability of some formulae in a reasonable time frame, it is force terminated and reports the result of unknown. It is worth noting that WyCS and the extension contain different forced termination criteria for handling large formulae. WyCS terminates based on the number of reduction or inference rules activated (introduced in § 6.1), while the extension terminates based on a time-out value. The limitations of the extension using a time-out value is discussed in § 5.1.2.

### 5.1.2 Limitations and Implications

Provided that a JUnit test does not throw an error, they may return one of two results: *pass* or *fail*. Unfortunately this was not enough to describe the three possible results of a verification tool: *satisfiable*, *unsatisfiable* or *unknown*. A result of unsatisfiable corresponded to a passed JUnit test and the results of satisfiable and unknown both corresponded to a failed JUnit test. The consequence of this was no adequate method for distinguishing—by just looking at the passed and failed statistics—whether the verification tool got an incorrect result (satisfiable) or was just unable to solve the problem before forced termination (unknown). It was possible however—given that there was only a small number of tests in the test suite—to look at individual tests to determine why they failed. This was valuable as the distinction between incorrect results and forced termination is important in comparing the performance of the verification tools.

WyCS and the extension use different forced termination criteria, introducing a limitation on some of the tests that terminated prematurely (i.e., returned unknown). A WyCS file contains multiple formulae to check, wherein WyCS resets its termination criteria when verifying each formula. The extension likewise translates a WyCS file into a single SMT-LIB file with multiple formulae. However, SMT-LIB only supports time-out criteria for entire files and not individual formulae. Thus, as the number of formulae increase, it becomes more likely that the extension would be forced to terminate prematurely. While this was a limitation with the extension (and only for the small subset of tests that returned unknown), we found that doubling the time-out value for the extension and re-running the evaluation had no effect on the results. Thus, it was unclear whether further increasing the time-out value or splitting the formulae up into separate files would actually help with verification.

| Verification Tool | Valid Tests ($n$ / 91) | Invalid Tests ($n$ / 42) |
|---|---|---|
| **WyCS** | 57 passed, 32 failed, 2 ignored | 42 passed |
| **Z3** | 63 passed, 28 failed | 22 passed, 20 failed |

**Table 5.1:** Test coverage comparison between WyCS and Z3 (WyCS JUnit test suite). The verification tools (WyCS and Z3) were run over the WyCS test suite. Two tests were ignored. The reason being that WyCS got stuck in an infinite loop, so were treated as failed tests.

### 5.1.3 Results

Table 5.1 shows the results of WyCS and Z3 on the WyCS test suite. To summarise, WyCS performed better on the invalid test group and Z3 on the valid test group.

WyCS passed all of the invalid tests while Z3 failed 20—due to returning unknown. This meant that Z3 was force terminated on these tests, as opposed to arriving at an incorrect result (§ 5.1.2).

WyCS ignored two valid tests as they caused WyCS to enter an infinite loop, and thus were treated as failed tests but not as incorrect results. In contrast, Z3 did not enter any infinite loops on the tests.

### 5.1.4 Discussion

The results of running WyCS and Z3 on the WyCS test suite showed that there was no clear best tool. Z3 did not surpass WyCS by a significant margin as it was hypothesised to do so. Instead it performed similarly to WyCS, corroborating the value of using an external SMT solver for the verification of Whiley programs. It may be concluded that the extension accurately translated the majority of WyCS features as Z3 was able to verify many of them. Furthermore, while Z3 had a number of failed tests (20 for the invalid test group), these failed tests were not due to incorrect results. This means Z3 may be used for the verification of Whiley programs as there were no *false negatives*.

An in-depth look at some of Z3's failed tests was taken. The majority of the failed tests utilised complex data structures such as sets or lists. This meant that multiple quantified conjectures were present, potentially making it difficult for Z3 to perform well (§ 4.1.3). It was hence theorised that reducing the number of quantified conjectures may have improved the results. One way of accomplishing this would have been to integrate at the WyAL level, which is discussed in detail later on as future work (§ 7.1).

An additional reason some of the tests failed was due to the omission of conjectures asserting the relation between the set subset, proper subset and length operators (§ 4.2.2). Despite their absence, they did not cause any false negatives, only some false positives. The use of the conjectures was experimented with; it was found that their presence allowed some tests to pass, but caused more to fail. Hence it was concluded that the absence of the conjectures would be better as it provided a higher success rate.

## 5.2 Experiment 2 – Performance over WyCS JUnit Tests

Experiment 1 evaluated the extension's accuracy in translating WyCS into SMT-LIB. This experiment was aimed at evaluating the extension's performance in terms of execution time. The experiment was performed by repeating Experiment 1 multiple times with each verification tool to calculate timed run statistics. It was performed over the WyCS test suite (as opposed to the Whiley test suite) in order to gain an accurate comparison of just the verification tools and not the whole compilation process.

### 5.2.1 Methodology

This experiment was run using the WyCS test suite. As our interest was in the comparison of WyCS's and the extension with Z3's verification abilities, only tests that were compilable were selected.

Calculating statistics over Java applications required care. Variables such as Just-in-time compilation, Java Virtual Machine (JVM) startup time and Garbage Collection (GC) runs needed to have their variance minimized in order to gain accurate performance statistics of the verification tools involved. Minimization of these variances was achieved by using performance evaluation techniques such as JVM warmup runs and averaging over multiple runs, as described by Georges *et al* [34].

If an experiment is run without taking JVM startup time into account then the results may be negatively skewed. The JVM startup time may be removed as a factor by performing warmup runs and excluding these runs from the final average. This experiment performed 20 warmup runs prior to the execution of each test. It was found that past 10 warmup runs the effect on the results was negligible, so 20 was a good number.

A variety of variables are uncontrollable during the execution of a Java program, such as GC runs. GC runs may cause a spike in execution time. In order to account for this and to reduce the impact on the results, multiple executions of each test were performed and averaged. The number of runs per test was 20. Furthermore, the standard deviation was calculated for each test to illustrate confidences.

### 5.2.2 Limitations and Implications

The experiment did not include any data about whether a test passed or failed. This could have been useful as a failed test was often caused by forced termination, i.e., a time-out. In these events, the reported execution time for a test was quite large and did not provide useful information. While one option would have been to remove all prematurely terminating tests, this would have lost some information about the verification tool that was not force terminated. Therefore the tests were left in the experiment but could easily be removed from the generated graphs later on.

The extension translates the WyCS file format into the SMT-LIB file format. It then writes out the SMT-LIB translation to a temporary file before calling the external SMT solver to verify it. This process involves input–output (IO) operations, which are more costly than memory operations. Thus, the extension has an increased performance overhead when compared with WyCS. Unfortunately it was not clear how to remove this overhead to have a clean comparison of WyCS and Z3.

### 5.2.3 Results

Figure 5.1 compares the performance of WyCS and Z3 on the valid test group, likewise Figure 5.2 on the invalid test group. The graphs show the tests along the horizontal axis and the mean execution time on the vertical axis. The tests are ordered by the execution time of Z3 as a visual aid.

On the valid test group it can be seen that the tools performed both similarly and consistently on the majority of tests. A number of the tests showed the tools having execution times of larger than 600ms. These can be attributed to the tests being force terminated due to the tool's inability to verify them. Thus, neither tool significantly outperformed the other.

The invalid test group showed both verification tools performing consistently across the tests, with WyCS performing better than Z3 often by more than a factor of 2. The main theory for this was due to the requirement for the extension to write out the SMT-LIB file

before calling Z3 to verify it. The cost of IO operations is expensive compared to memory operations. While WyCS outperformed Z3, it was only by 25ms in the majority of cases which is negligible.

### 5.2.4 Discussion

The first positive conclusion that may be drawn from this experiment is that both tools are consistent across the majority of tests. This is important as it is undesirable to have the execution time of the Whiley compiler collection drastically increased by the use of verification.

It was hypothesised that Z3 would verify the tests faster than WyCS as it is a professionally developed tool, yet that was not the case as seen from this experiment. One theory for this was the fact that the extension had to write out the SMT-LIB file before calling Z3 to verify it. The cost of IO operations is expensive compared to memory operations. This would have caused an increase in the overhead for each call to run Z3. Another possible reason was due to the need to use quantifiers for encoding the WyCS data types in SMT-LIB; quantifier instantiation is an expensive process. Despite this result though, the difference between the tests is negligible in terms of consequence to the Whiley compiler collection. This experiment showed that using Z3 instead of WyCS would not improve performance, but neither would it significantly hinder it.

## 5.3 Experiment 3 – Whiley JUnit Tests

The Whiley JUnit tests are written in Whiley. They use the native language features of Whiley such as sets and maps. For each individual test, the test must be compiled using the Whiley compiler into the WyIL, WyAL, WyCS and SMT-LIB file formats before verification. This means the tests are testing the compiler collection as a whole and are more representative of the usage scenarios for the extension. Consequently, they are testing the ability of the extension and external SMT solver to verify representative and complex translations of Whiley programs.

### 5.3.1 Methodology

This experiment was the same as Experiment 1 (§ 5.1.1) with the following one difference: the Whiley test suite was used instead of the WyCS test suite.

### 5.3.2 Limitations and Implications

The limitations from Experiment 1 (§ 5.1.2) applied to this experiment. To recap, these limitations were the following: the inability to distinguish between an incorrect result and forced termination for a failed test; and the usage of different termination criteria (which only affected tests that were force terminated).

The use of the Whiley test suite introduced one further limitation; not all of the tests in the test suite use verification. This limitation only affected how the results from this experiment may be reported, not the results themselves. To elaborate by example, it is invalid to claim *Z3 failed to verify 31.9% of the tests* as some tests failed due to compiler errors rather than verification errors. It is however valid to claim that *Z3 performed worse than WyCS* and that *the 77 failed tests can be attributed to the use of Z3 instead of WyCS*. This limitation was not present in Experiment 1 as the tests that failed due to compiler errors were manually removed. This experiment however contained almost 1000 tests, which made it infeasible to manually remove them.

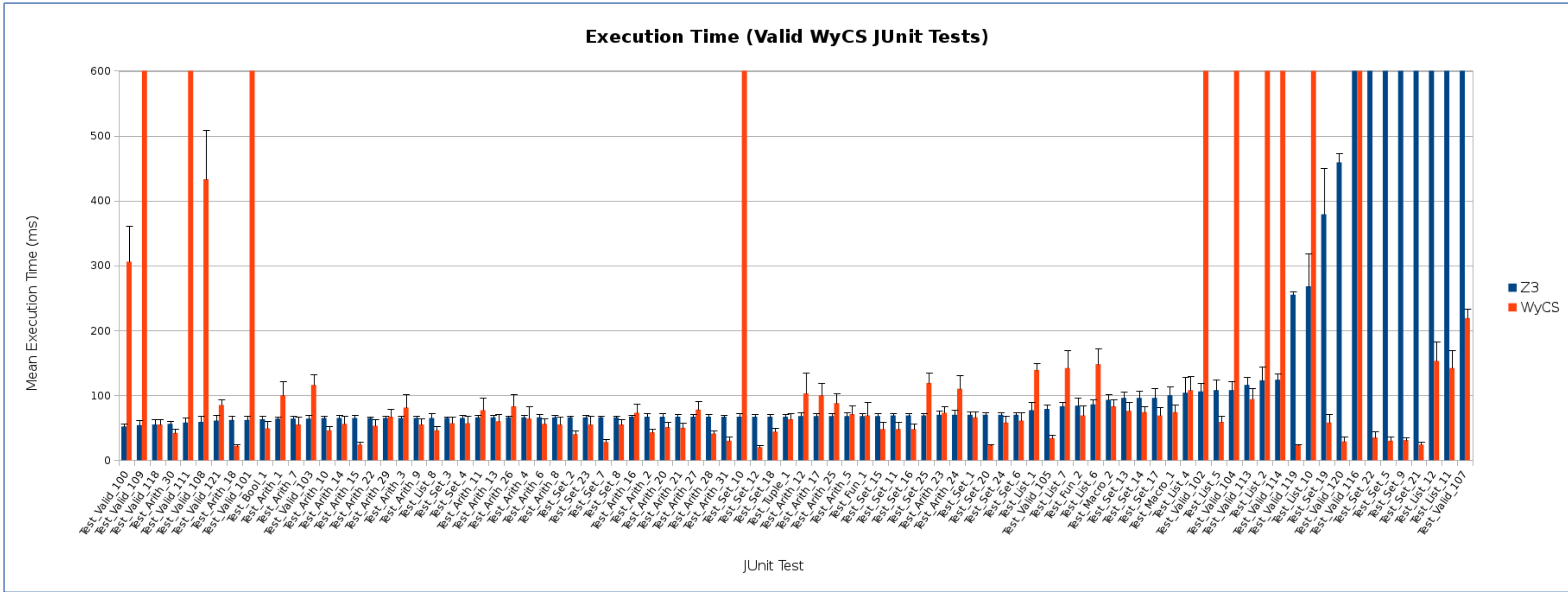


**Figure 5.1:** Execution time comparison between WyCS and Z3 (valid WyCS JUnit tests). This bar graph shows and compares the mean execution time for WyCS and Z3 on the valid WyCS JUnit tests. The horizontal axis lists each test, ordered by Z3's performance to visually aid interpretation. The vertical axis is the mean execution time of each test (over 20 runs) in milliseconds. The error bars show one standard deviation. A number of tests had mean execution times larger than the maximum unit on the vertical axis, due to them timing out and hence it was not considered useful to scale the axis to them.

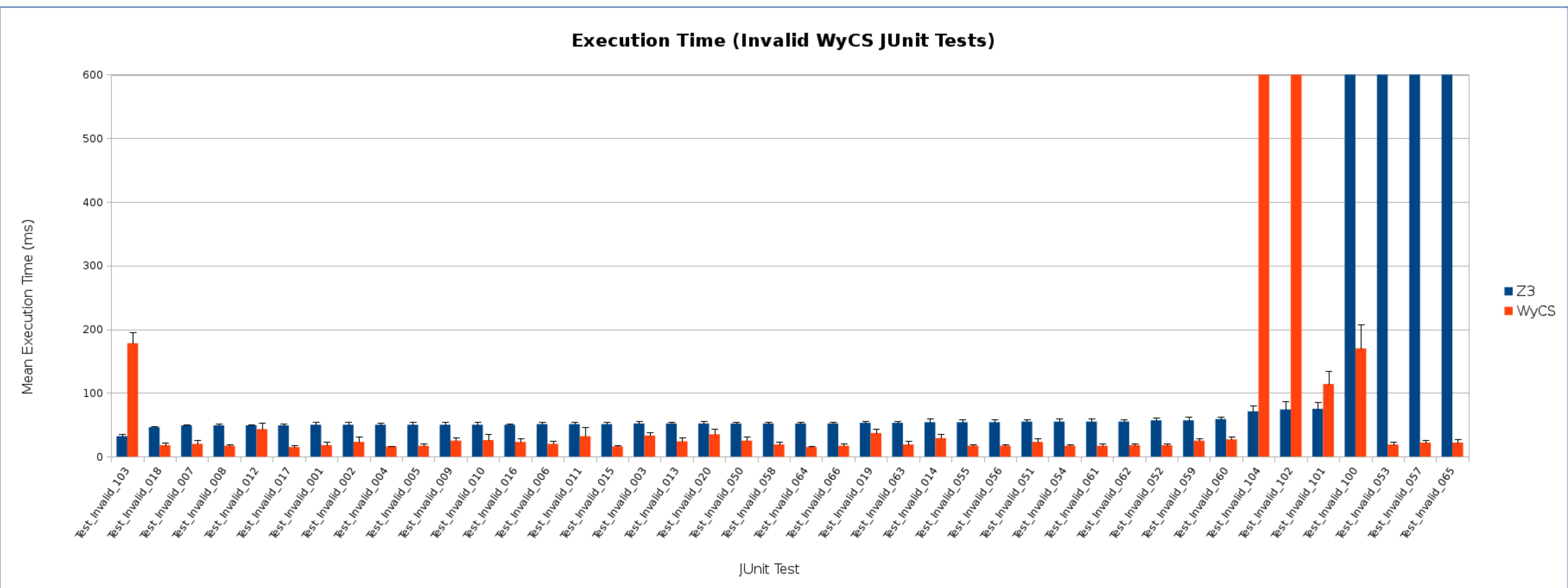


**Figure 5.2:** Execution time comparison between WyCS and Z3 (invalid WyCS JUnit tests). This bar graph shows and compares the mean execution time for WyCS and Z3 on the invalid WyCS JUnit tests. The horizontal axis lists each test, ordered by Z3's performance to visually aid interpretation. The vertical axis is the mean execution time of each test (over 20 runs) in milliseconds. The error bars show one standard deviation. A number of tests had mean execution times larger than the maximum unit on the vertical axis, due to them timing out and hence it was not considered useful to scale the axis to them.

| Verification Tool | Valid Tests ($n$ / 627) | Invalid Tests ($n$ / 287) |
|---|---|---|
| **WyCS** | 455 passed, 137 failed, 35 ignored | 244 passed, 43 ignored |
| **Z3** | 433 passed, 163 failed, 31 ignored | 189 passed, 55 failed, 43 ignored |

**Table 5.2:** Test coverage comparison between WyCS and Z3 (Whiley JUnit test suite). The verification tools (WyCS and Z3) were run over the Whiley test suite. Some tests were ignored. The reason for this was that the execution got stuck in an infinite loop, so were treated as *failed* tests.

### 5.3.3 Results

Table 5.2 shows the results of WyCS and Z3 on the Whiley test suite. WyCS performed better than Z3 across both test groups (valid and invalid).

WyCS had 22 and 55 additional passed tests on the valid and invalid test groups respectively. Combined, that was 8.4% better than Z3. While this percentage is small, 77 failed tests for Z3 is significant.

### 5.3.4 Discussion

The statistics recorded in Table 5.2 show that WyCS performed better than Z3 for the verification of Whiley programs. However as explained in § 5.3.2, it is not valid to make claims on how much better WyCS performed. Experiment 4 in § 5.4 evaluates the Whiley benchmarks and would allow such claims to be made as the benchmarks are known to use verification.

A sample of failed tests for Z3 were looked at in detail to determine their reason for failure. It was found that many failed due to timing out. These tests were re-run with double the time-out value to ascertain whether increasing the time-out would help their verification. The re-runs all still failed due to timing out, suggesting that the time limitation may not have been the cause of their failure.

## 5.4 Experiment 4 – Whiley Benchmarks

The Whiley benchmarks are micro to medium-sized programs written in Whiley, ranging from simple mathematical functions such as Fibonacci to complex programs such as Chess. Some of the benchmarks facilitate verification, such as *006_queens*. These benchmarks provided another method for comparing WyCS and the extension with Z3. This experiment provided a more accurate test of the extension's verification abilities on representative and complex translations of Whiley programs than Experiment 3 (§ 5.3).

### 5.4.1 Methodology

The suite contains over 30 micro, small- and medium-sized benchmarks. In order to have a statistically unbiased experiment, fair selection criteria were required. A benchmark was selected for this experiment if it was a) of a micro or small size, b) compilable and c) designed for verification. A benchmark was considered designed for verification if it contained any `requires`, `ensures` or `where` keywords as they are directives for the Verification Condition Generator.

This experiment was run later than Experiments 1, 2 and 3. The version of Whiley and WyCS had been updated to 0.3.29. Z3's version (4.1) remained the same.

The two verification tools were run over the selected benchmarks and the verification result was recorded. The configuration options for WyCS and the extension remained the same as in Experiment 1 (Appendix A).

### 5.4.2 Limitations and Implications

This experiment used a sample of the Whiley benchmarks as described above and thus, only a sample of Whiley language features were tested. This is a limitation on the experiment as it may have omitted some of Whiley's key language features. In order to improve the experiment for future evaluations, additional benchmarks should be included. This would involve ensuring more benchmarks compile and are correctly designed for verification.

The definition of *designed for verification* in the selection criteria was quite loose. The definition did not guarantee the benchmark to be verifiable, e.g., some of the pre- and post-conditions may have been invalid. This limitation was not taken into account when the results were reported and so a result of *verification unsuccessful* did not necessarily mean the result was incorrect. The verification tool may be correct in that the given benchmark was unverifiable. Designing a program correctly for verification is a challenge in itself (e.g., developing loop invariants [37, 31]) and was considered out of scope of this experiment.

### 5.4.3 Results

Table 5.3 shows the results of WyCS and Z3 on the selected benchmarks. WyCS successfully verified 3 out of 12 of the benchmarks and reported 4 as unverifiable. Z3 only verified 2 out of 12.

A large number of the benchmarks Z3 failed to verify were due to time-outs. When re-running Z3 without a time-out value however, it reported function overloading ambiguities. Recall that utility functions (e.g., set `add`) had to be overloaded for each sort interpretation (§ 4.1.2 and § 4.1.3). Thus, function overloading ambiguities refers to the inability of Z3 to determine which function to call.

### 5.4.4 Discussion

The types of the failures (time-out, unknown, function ambiguities) highlight interesting differences between WyCS and the extension. Failures of function ambiguities showed that the translation process is difficult. While Z3 supports function overloading, this experiment showed that Z3 has difficulties in deducing which function to call.

A closer look at the benchmarks that failed due to function ambiguities was taken. Unfortunately, it was unclear exactly why the error was occuring. However, we theorised that it was potentially caused by either an error in the translation process or by Z3. Name mangling the utility function definitions (e.g., `get` or `length`) as opposed to overloading them may have helped to solve the issue.

An in-depth look at the rest of the benchmarks for the extension was taken. It was found that the verified benchmarks used only simple data types (e.g., integers and booleans) and no complex data types (e.g., sets). This showed that the extension is able to verify simple Whiley programs. The benchmarks that the extension failed to verify contained either more complex data types or a large quantity of assertions. This showed that the extension has difficulty with complex Whiley programs.

| Benchmark | Whiley | Z3 |
|---|---|---|
| **002_fib** | Successfully verified | Returned unknown |
| **004_matrix** | Verification unsuccessful | Reported function overloading ambiguities |
| **006_queens** | Timed-out | Reported function overloading ambiguities |
| **007_regex** | Timed-out | Reported function overloading ambiguities |
| **009_lz77** | Verification unsuccessful | Reported function overloading ambiguities |
| **010_sort** | Verification unsuccessful | Returned unknown |
| **012_cyclic** | Timed-out | Timed-out |
| **019_subseq** | Timed-out | Timed-out |
| **022_cars** | Successfully verifies | Successfully verifies |
| **023_microwave** | Successfully verifies | Successfully verifies |
| **024_bits** | Verification unsuccessful | Timed-out |
| **102_conway** | Timed-out | Timed-out |

**Table 5.3:** Performance comparison between WyCS and Z3 (Whiley benchmarks). The verification tools (WyCS and Z3) were run over a sample of Whiley benchmarks. Each benchmark compiled, so the table reports the result of the verification attempt for each tool. The result *timed-out* refers to forced termination of the verification tool, e.g., for WyCS it means the maximum number of rewrite rules was reached.

## 5.5 Summary

Experiment 1 showed that WyCS and the extension with Z3 performed similarly on the WyCS JUnit test suite. This validated the idea of using an external SMT solver to verify Whiley programs by showing that it is possible to accurately encode the WyCS file format in SMT-LIB. The experiment further highlighted issues with individual WyCS language features or encodings. Specifically, some encodings introduced too many quantifiers and made it difficult for Z3 to perform verification. It was hypothesised that a solution to this limitation would have been to integrate at the WyAL level instead of WyCS (discussed as future work in § 7.1).

Experiment 2 further evaluated WyCS and the extension with Z3 on the WyCS JUnit test suite. It showed that Z3 took longer to execute but that the difference to WyCS was negligible when considering it as part of the whole compilation process.

Experiment 3 showed that WyCS performed better than the extension with Z3 on the Whiley JUnit test suite. However due to the experiment's limitations it was not possible to make claims regarding how much the performance differed.

Experiment 4 showed that the extension with Z3 is able to verify simple Whiley programs. This was promising as it illustrated that despite some encodings of complex data types (lists, maps, etc.), the extension is able to perform the verification step in the Whiley compilation process. It further illuminated areas of future work for this project, e.g., removing the function overloading ambiguities.

Each experiment was completed using Z3 as the external SMT solver with the extension. This restricts some of the conclusions to this domain. Further experiments with alternative SMT solvers would help to solidify the conclusions made from the experiments.

# Chapter 6

# Additional Experiment

This chapter introduces the inner workings of WyCS and an additional experiment that analysed the most frequently used rewrite rules. WyCS is the SMT solver developed by David J. Pearce as part of the Whiley project. It takes a list of formulae and attempts to prove their satisfiability or unsatisfiability by using theorem proving techniques. Rewrite rules are a key part of manipulating the formulae in order to decide their satisfiability. They come in two forms, *reduction rules* and *inference rules*. Reduction rules are used to rewrite formulae into simplified or normalised equivalents while inference rules are used to infer new formulae given some existing formulae. More on the background and theory of this can be found below.

During verification there is a choice about a) which rewrite rules to apply b) to which formulae and c) in what order. Each of these decisions can affect the performance of WyCS. This experiment evaluates the usage of WyCS's rewrite rules in order to gain a greater understanding of their individual importance, specifically, what is the expected probability of a rewrite rule *activating* given a random test. A rule activation refers to each attempt of WyCS to use the rule (either successfully or unsuccessfully) on some formulae. A rule is defined as *used* if it is activated at least once during the test. The aim was to enable stronger justifications of rewrite rule selection criteria and also to indicate which rules should be optimised to best improve WyCS's performance.

## 6.1 WyCS Architecture

This section provides the background information about WyCS necessary to understand this experiment. The field of theorem proving is vast and complex and as such, a full and comprehensive guide is out of this project's scope.

WyCS has a *search space* which contains a list of the currently active formulae. WyCS further has a *root formula* and a list of rewrite rules, both for reductions and inferences. In order to apply a rewrite rule, an attempt is made to pattern match it against either the root formula or some subformulae of it. As mentioned above, reduction rules attempt to simplify or normalise some formulae, potentially making the search space smaller. Inference rules attempt to infer (create) new formulae from some existing formulae, making the search space larger.

As an example to explain this concept, take the following root formula where `a` and `b` are booleans and `x`, `y` and `z` are integers: $(a \wedge b) \vee (x \geq y \wedge \neg(\neg(y \geq z)))$. One reduction rule is the *Not_2* rule, defined as $\neg(\neg x) \rightarrow x$. This rule takes a double negation of any term and eliminates the negations. WyCS could try to pattern match the rule against the root formula, where it would find that it does not match the root formula itself, but does

match a subformula. After application of the reduction rule, the root formula would become $(a \wedge b) \vee (x \geq y \wedge y \geq z)$. A key idea here is that the formula was simplified, i.e., the formula has not increased in size. An inference rule on the other hand takes some formulae and infers some new formulae. An example inference rule is transitivity of inequalities: $a \geq b \wedge b \geq c \rightarrow a \geq c$. Application of this rule to our reduced root formula would yield $(a \wedge b) \vee (x \geq y \wedge y \geq z \wedge x \geq z)$. Note how the formula has increased in size.

From the above example, it can be seen how the order of rule application affects the performance of WyCS; the transitivity of inequalities rule would not have matched the formula if applied before the negation elimination. Hence, it would have required an attempt to match the rule twice rather than once.

The verification process, while convergent (i.e., will terminate), may take an impractical amount of time. Therefore, WyCS has a maximum cap on the number of rewrite rules that may be applied before it forces premature termination. The default cap for WyCS is 500 reduction rules and 200 inference rules. Therefore efficient selection criteria for rewrite rules is a critical component of WyCS's verification process.

WyCS has three rewrite modes, *simple*, *static dispatch* (the default) and *global dispatch*. Simple works by iterating through each formula in the search space and trying to apply every rewrite rule to it. Static dispatch is the default and is similar to simple, except that it only attempts to apply a rewrite rule to a formula if their types match. In this mode, a hash of types to applicable rules is pre-computed. Global dispatch follows a similar method to static dispatch, except that it iterates through each rewrite rule and attempts to apply each to every formula (in contrast to static dispatch where it iterates through each formula and attempts to apply every rewrite rule to it). The iteration order of the rewrite rules could be sorted by the most probable rule to be used or the most beneficial rule in terms of reducing the formulae. The WyCS rewrite modes lack this kind of knowledge about the rewrite rules, which is where this experiment comes in.

## 6.2 Experiment 5 – WyCS Rewrite Rules

This experiment answered the following question: what is the expected probability of a rewrite rule activating given a random test? The experiment evaluated the rewrite rule activation probabilities over the WyCS test suite. The WyCS test suite was used as it tests the different language features of WyCS. We believed it would provide the broadest range of scenarios in which different rewrite rules might be activated. The aim of this experiment was to enable more informed decisions regarding rewrite rule selection criteria and also to highlight which rules might provide the most performance benefit if optimised.

### 6.2.1 Methodology

Experiments 1, 2 and 3 split the tests up based on the test suite's valid and invalid groups (recall Chapter 5). This experiment split the tests up based on whether they passed or failed. Recall that a failed test means that WyCS either got the wrong answer or the forced termination criteria was met. Further recall that WyCS's termination criteria is the number of rewrite rule activations. It was theorised that the failed tests would thus have higher activation counts than the passed tests, hence the reason for separating the test suite up based on whether the tests passed or failed. Additionally, any tests that failed due to compiler issues were removed as they would provide no insight into the rewrite rule activations of WyCS.

Whiley and WyCS were at version 0.3.27 at the time of this experiment. The configuration options for WyCS followed as in Appendix A (omitting the irrelevant extension configuration options).

WyCS was run over each compilable test in the passed and failed groups and the rewrite rule activation counts were recorded. The following two statistical analyses for each rewrite rule were then conducted:

1. **Usage percentage**

   The usage percentage is termed as the percentage of tests in which a rewrite rule was activated at least once. This statistic was useful for determining how widely used a rule was across all of the tests. Furthermore, this statistic allows informed decisions about which rules should be optimised.

2. **Expected activation probability**

   The expected activation probability given a random test was calculated for each rewrite rule. The tests in each test group were uniformly distributed (i.e., each test had the same chance of occurring) and so the expected value calculation was equivalent to the mean of the rewrite rule's activation probabilities. The activation probability of a rule in any given test was just its number of activations divided by the total number of rule activations. The expected activation probability allows informed decisions regarding which rules should be prioritised and optimised in WyCS.

### 6.2.2 Limitations and Implications

This experiment provided a method for calculating the expected activation probability of a rewrite rule given a random WyCS JUnit test. Thus, the conclusions are restricted to this domain; the expected activation probability of a rewrite rule on actual Whiley programs may be different. While this is a restriction on the use of the results, it is theorised that the activation probability of a rewrite rule given a random Whiley program would be similar. Further experiments over a sample of Whiley programs could validate this theory and solidify the conclusions drawn.

This experiment did not relate the usage percentage or expected activation probability to the WyCS language features. While not a limitation on the experiment itself, it would have provided a useful statistic. The statistic could allow for a rewrite rule heuristic that favoured rules dependent upon the language features used in the formulae.

### 6.2.3 Results

Figures 6.1 and 6.2 show the usage percentage of the rewrite rules for the passed and failed tests. The horizontal axis shows the rule and the vertical axis the usage percentage. The usage percentage is the proportion of tests in which each rule was activated at least once. Interestingly, both graphs show a non-uniform usage of rules; some are used in many tests while others in near none. Furthermore, while there was a correlation between the rules from the passed and failed test groups, the most used rewrite rule differed (*Multiplication_2* and *Sum_4* for the passed and failed test groups respectively).

Figures 6.3 and 6.4 show the expected activation probability of the rewrite rules for the passed and failed tests. The horizontal axis shows the rule and the vertical axis the expected activation probability. Each rule has positive error bars, showing one standard deviation. The majority of rules did not have an expected activation probability of more than 10%. The most likely rule to be activated given a random test was the same for both test groups, *Sum_4*.

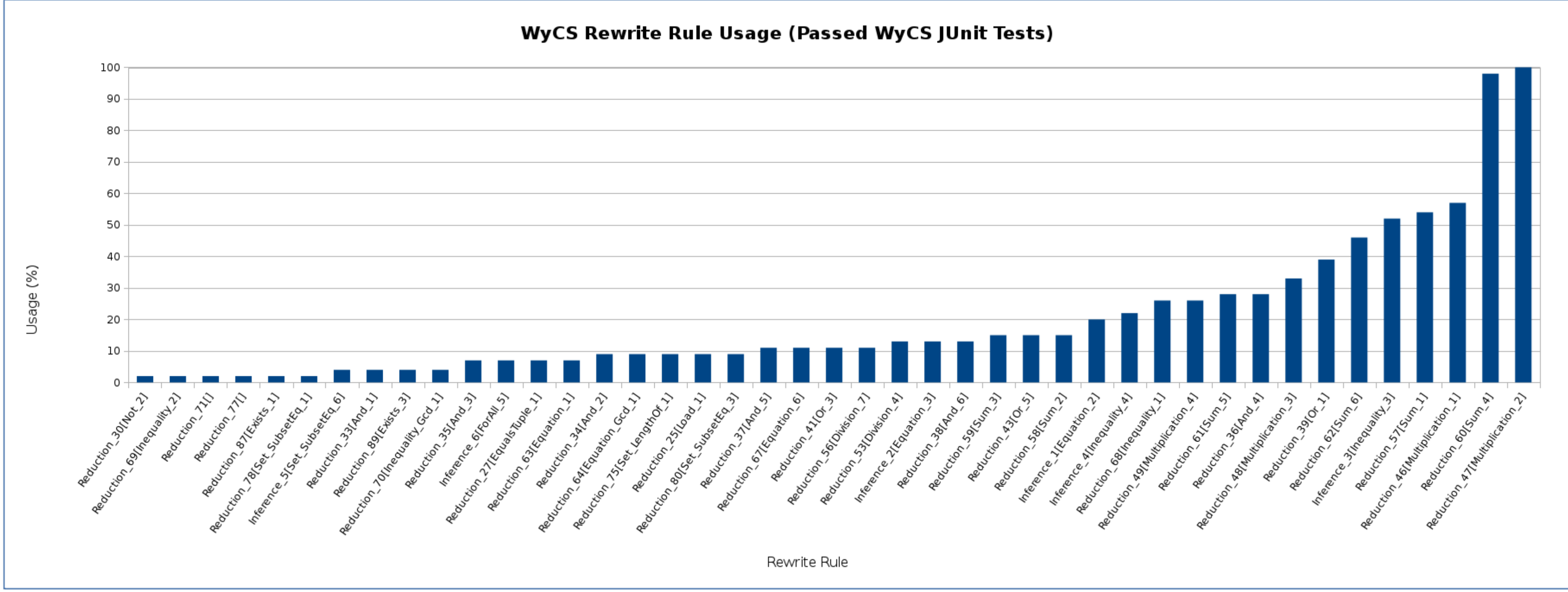


**Figure 6.1:** WyCS rewrite rule usage (passed WyCS JUnit tests). This bar graph shows and ranks the WyCS rewrite rule usages for the passed WyCS JUnit tests. The usage of a rewrite rule is the percentage of passed JUnit tests in which it is activated at least once. The horizontal axis lists each rewrite rule, with the least used on the left-hand-side and the most used on the right-hand-side.

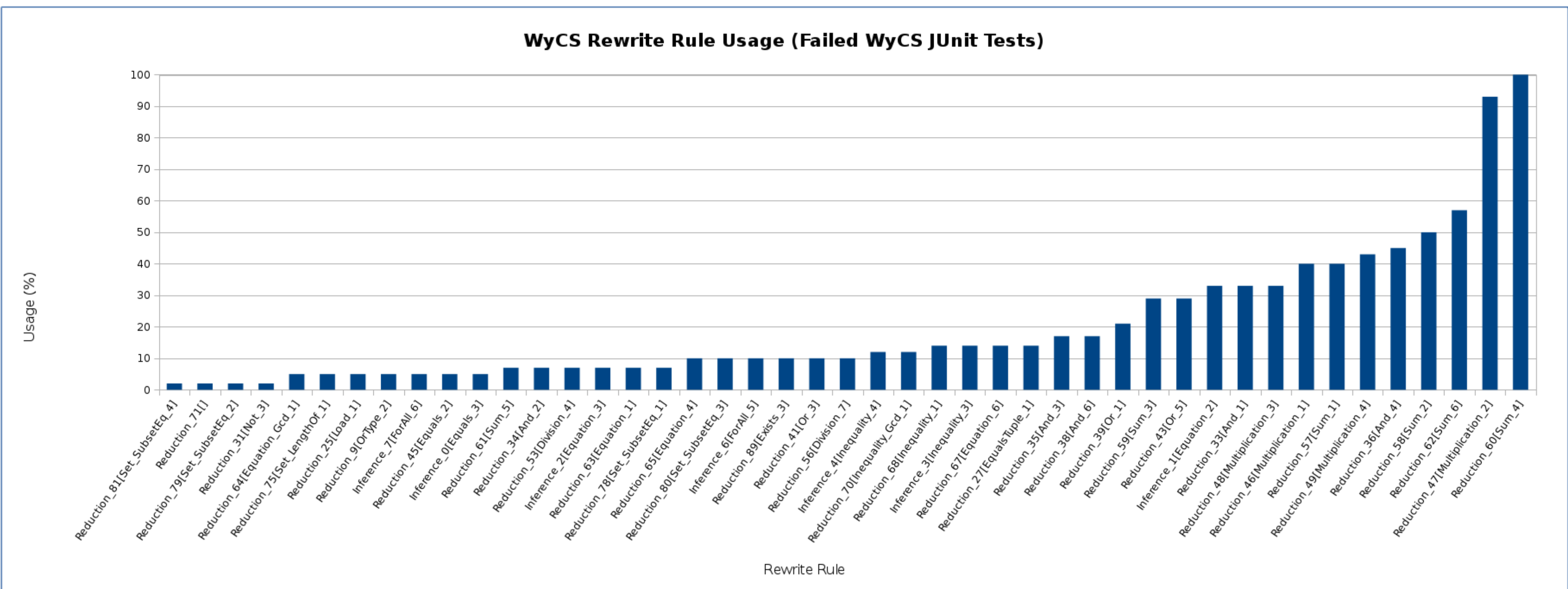


**Figure 6.2:** WyCS rewrite rule usage (failed WyCS JUnit tests). This bar graph shows and ranks the WyCS rewrite rule usages for the failed WyCS JUnit tests. The usage of a rewrite rule is the percentage of failed JUnit tests in which it is activated at least once. The horizontal axis lists each rewrite rule, with the least used on the left-hand-side and the most used on the right-hand-side.

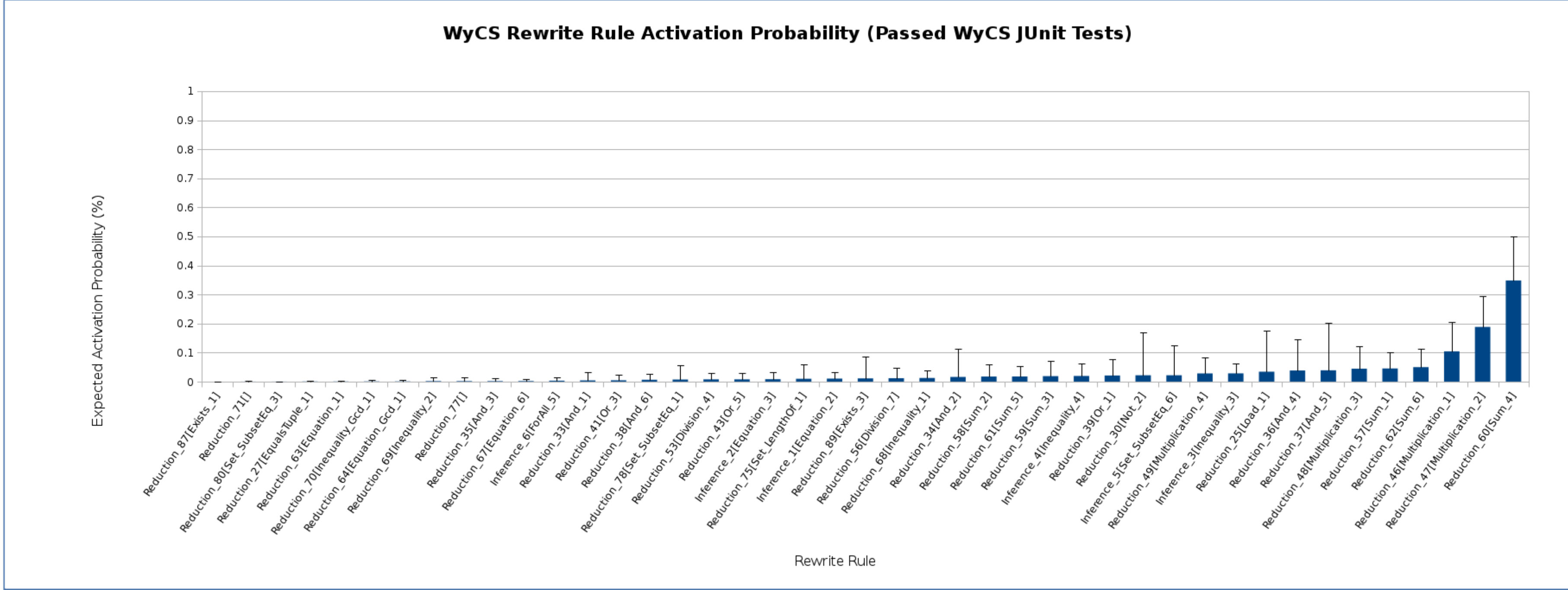


**Figure 6.3:** WyCS rewrite rule activation probability (passed JUnit tests). This bar graph shows and ranks the WyCS rewrite rules' expected activation probabilities for the passed WyCS JUnit tests. As the passed tests were uniform (had an even chance of occurring), the expected activation probability for a single rewrite rule was equivalent to the mean of its activation probabilities for all tests. The horizontal axis lists each rewrite rule, with the least likely to be activated on the left-hand-side and the most likely to be activated on the right-hand-side. The error bars show one standard deviation.

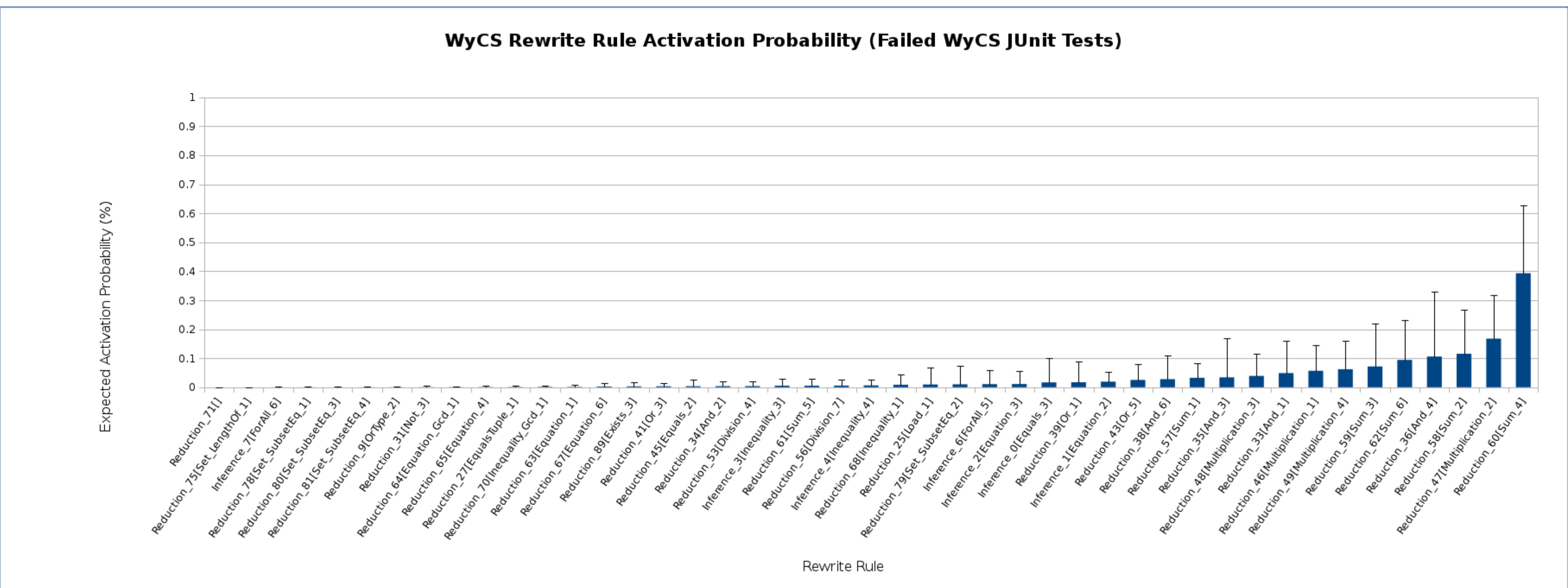


**Figure 6.4:** WyCS rewrite rule activation probability (failed WyCS JUnit tests). This bar graph shows and ranks the WyCS rewrite rules' expected activation probabilities for the failed WyCS JUnit tests. As the failed tests were uniform (had an even chance of occurring), the expected activation probability for a single rewrite rule was equivalent to the mean of its activation probabilities for all tests. The horizontal axis lists each rewrite rule, with the least likely to be activated on the left-hand-side and the most likely to be activated on the right-hand-side. The error bars show one standard deviation.

### 6.2.4 Discussion

The rewrite rule usage percentage varied greatly for both the passed and failed test groups. Of interest was the fact that the most used rules pertained to arithmetic operations, e.g., *Sum_4*, *Sum_6*, *Multiplication_1* and *Multiplication_2*. It was initially thought that the most used rules would relate to boolean operations, e.g., *And_4* and *Or_1*. This was because all formulae use predicate logic and hence could utilise the boolean rewrite rules to help decide their satisfiability. This potentially indicated that the formulae were already written in a normalised form in the tests themselves. Repeating this test for the Whiley test suite may yield different results.

The expected activation probability of the rewrite rules showed that *Sum_4*, *Multiplication_2* and *Multiplication_1* were the most likely to be activated given a random passed test. *Sum_4*, *Multiplication_2* and *Sum_2* were the most likely to be activated given a random failed test. The types of these rewrite rules are similar to the most used rewrite rules, that is, they all pertain to arithmetic operations. This statistic was unforeseen; the tests utilised a fair amount of other data types (e.g., sets and lists) yet these data types' rewrite rules (*Set_SubsetEq_1*, *Set_LengthOf_1*, etc.) were not activated a large number of times. Based on these results showing that arithmetic-based rewrite rules are the most likely to be activated, it is concluded that the WyCS's rewrite rule selection criteria would benefit from prioritising their activation and optimising them.

Clearly from both analyses, *Sum_4* is one of the most used and most likely to be activated rewrite rules. *Sum_4* reduces a summation expression that contains at least two constants down to one with one constant. In hindsight, it is not surprising that this rewrite rule came as one of the most used and most likely to be activated. This result suggests that any improvement to the implementation and efficiency of this rule could significantly aid the verification process. However, this rule is one of the more simple ones and actual efficiency improvements may be difficult to implement.

# Chapter 7

# Future Work and Conclusions

This chapter discusses the future work beyond this project and the conclusions drawn from the project as a whole.

## 7.1 Future Work

This project has laid the groundwork and more for using external SMT solvers for the verification of Whiley programs. The usage of SMT solvers has been validated as a hypothesis, but further work could be done to improve their performance in this area. Three main areas in particular were illuminated for future work throughout this project:

- **The integration point for the extension (discussed in § 3.2).** This project integrated at the WyCS level but integrating at the WyAL level may have alleviated some of the issues encountered during implementation and evaluation. The main issue was that integrating at WyCS meant there was a double encoding of some data types (i.e., lists and maps) in SMT-LIB. This meant there were more quantified conjectures for these encodings than would have been necessary if the extension integrated at the WyAL level.

- **Additional Whiley language features.** Whiley supports many advanced language features that are not yet supported by WyAL, WyCS and consequently, the extension. Once WyAL and WyCS support more of Whiley's advanced language features (as they are planned to do), the extension will require additions to incorporate them. The language features foreseen to be required include recursive functions, composite types (e.g., unions) and subtyping relations. The incorporation of these language features presents an interesting challenge as the SMT-LIB file format does not lend itself to features of their nature.

- **Alternative SMT solvers such as CVC4.** The extension was chosen to target SMT-LIB in order to decouple it from any one external SMT solver. Unfortunately—while unplanned—the use of Z3 throughout the project had an effect on the implementation of the extension. In order to utilise alternative SMT solvers, small modifications may be required. For example, the extension overloads some functions (e.g., `get` and `length`) but some SMT solvers such as CVC4 do not support function overloading. Using name mangling instead of function overloading is one possible solution here.

## 7.2 Conclusions

To recap from § 1.1, this project contributed the following:

1. A method for using external SMT solvers for the verification of contracts in Whiley programs (Chapters 3 and 4).
2. An evaluation of the above method using the existing Whiley test and benchmark suites (Chapter 5).
3. An analysis of the most frequently used and most likely activated rewrite rules in WyCS, allowing for informed future development of rewrite rule selection criteria (Chapter 6).

The design and implementation of the extension (1) was successful; the evaluation of the extension (2) validated the hypothesis of using external SMT solvers for the verification of Whiley programs. The evaluation further showed that the extension outperformed WyCS on simple language features (booleans, integers, etc.) but lacked the ability to verify complex language features (sets, lists, etc.). An in-depth look was taken and it was identified that integrating the extension at the WyAL level instead of WyCS may alleviate this issue.

The design of the extension was modular and largely decoupled from any one external SMT solver, owing to it targeting the SMT-LIB standard language specification. This laid the groundwork for verifying Whiley with any external SMT solver, provided it supports SMT-LIB. The implication of this is vast; for example, it could allow for:

- The future development of solver-specific extensions for performance or language improvements.
- The (concurrent) use of multiple SMT solvers for the verification of Whiley programs.
- The use of the SMT-LIB benchmark suite to further evaluate WyCS and the extension.

This project additionally provided an analysis of WyCS's rewrite rule activations (3), specifically, their expected probability. The statistics generated from this analysis showed that given a random WyCS test, the most likely rewrite rule to be activated is arithmetic-based (e.g., *Sum_4*). This information may be used to educate the WyCS rewrite rule selection criteria. Furthermore, it may be used to selectively optimise individual rewrite rules to improve WyCS's performance. While the statistics were restricted to the domain of the WyCS test suite, it was theorised that there would be a correlation to Whiley programs and that the conclusions would remain valid. To validate this theory, our experiment could be re-run over the Whiley test suite or benchmarks.

Overall, the development of the extension for using external SMT solvers with Whiley was successful. Some interesting issues were encountered throughout this project and while some were overcome, others lead to potential future work (§ 7.1) that would further improve the extension.

# Appendices

# Appendix A

# Evaluation Configuration Options

This appendix details the configuration options used in the evaluation (Chapter 5). Two configurations are detailed here, one for each verification tool being evaluated (WyCS and the extension).

- **WyCS**

  **`debug`** = `false`

  **`rwMode`** = `STATICDISPATCH`
  : This option sets the rewrite mode of WyCS. Other modes available included `SIMPLE` and `GLOBALDISPATCH`.

  **`maxReductions`** = `500`
  : This option sets how many reduction rules may be activated before WyCS stops and returns unknown.

  **`maxInferences`** = `200`
  : This option sets how many inference rules may be activated before WyCS stops and returns unknown.

- **Extension**

  **`debug`** = `false`

  **`solver`** = `Z3`
  : This option sets the target solver to use.

  **`timeout`** = `10`
  : This option sets how long in *timeout-unit* until the extension stops the external SMT solver and returns unknown.

  **`timeout-unit`** = `SECONDS`
  : This option sets the timeout unit.

# Bibliography


[1] ABRIAL, J.-R. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[2] ABRIAL, J.-R., BUTLER, M., HALLERSTEDE, S., HOANG, T. S., MEHTA, F., AND VOISIN, L. Rodin: an open toolset for modelling and reasoning in event-b. *International journal on software tools for technology transfer 12*, 6 (2010), 447–466.

[3] ABRIAL, J.-R., AND HALLERSTEDE, S. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundamenta Informaticae 77*, 1 (2007), 1–28.

[4] AYEWAH, N., PUGH, W., MORGENTHALER, J. D., PENIX, J., AND ZHOU, Y. Using findbugs on production software. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion* (2007), ACM, pp. 805–806.

[5] BARNETT, M., CHANG, B.-Y. E., DELINE, R., JACOBS, B., AND LEINO, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects* (2006), Springer, pp. 364–387.

[6] BARNETT, M., FÄHNDRICH, M., LEINO, K. R. M., MÜLLER, P., SCHULTE, W., AND VENTER, H. Specification and verification: the spec# experience. *Communications of the ACM 54*, 6 (2011), 81–91.

[7] BARNETT, M., LEINO, K. R. M., AND SCHULTE, W. The spec# programming system: An overview. In *Construction and analysis of safe, secure, and interoperable smart devices*. Springer, 2005, pp. 49–69.

[8] BARRETT, C., CONWAY, C. L., DETERS, M., HADAREAN, L., JOVANOVIĆ, D., KING, T., REYNOLDS, A., AND TINELLI, C. Cvc4. In *Computer aided verification* (2011), Springer, pp. 171–177.

[9] BARRETT, C., DE MOURA, L., AND STUMP, A. Smt-comp: Satisfiability modulo theories competition. In *Computer Aided Verification* (2005), Springer, pp. 20–23.

[10] BARRETT, C., DETERS, M., DE MOURA, L., OLIVERAS, A., AND STUMP, A. 6 years of smt-comp. *Journal of automated reasoning 50*, 3 (2013), 243–277.

[11] BARRETT, C., STUMP, A., AND TINELLI, C. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)* (2010), vol. 13, p. 14.

[12] BARRETT, C., STUMP, A., AND TINELLI, C. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.smt-lib.org/`, 2010.

[13] BARRETT, C., AND TINELLI, C. Cvc3. In *Computer Aided Verification* (2007), Springer, pp. 298–302.

[14] BARRETT, C. W., SEBASTIANI, R., SESHIA, S. A., AND TINELLI, C. Satisfiability modulo theories. *Handbook of satisfiability 185* (2009), 825–885.

[15] BLACKBURN, M., BUSSER, R., NAUMAN, A., KNICKERBOCKER, R., AND KASUDA, R. Mars polar lander fault identification using model-based testing. In *Software Engineering Workshop, 2001. Proceedings. 26th Annual NASA Goddard* (2001), IEEE, pp. 128–135.

[16] BLAIR, M., OBENSKI, S., AND BRIDICKAS, P. Patriot missile defense: Software problem led to system failure at dhahran. Tech. rep., Technical Report GAO/IMTEC-92-26, United States-General Accounting Office-Information Management and Technology Division, 1992.

[17] BOLIGNANO, D. Integrating proof-based and model-checking techniques for the formal verification of cryptographic protocols. In *Computer Aided Verification* (1998), Springer, pp. 77–87.

[18] BRADLEY, A. R., AND MANNA, Z. *The calculus of computation*. Springer, 2007.

[19] CHALIN, P., AND JAMES, P. R. Non-null references by default in java: Alleviating the nullity annotation burden. In *ECOOP 2007–Object-Oriented Programming*. Springer, 2007, pp. 227–247.

[20] COK, D. R. The smt-libv2 language and tools: A tutorial. Tech. rep., Technical report, GrammaTech, Inc., Version 1.1, 2011.

[21] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[22] DE MOURA, L., AND BJØRNER, N. Satisfiability modulo theories: introduction and applications. *Communications of the ACM 54*, 9 (2011), 69–77.

[23] DELINE, R., AND LEINO, K. R. M. Boogiepl: A typed procedural language for checking object-oriented programs. Tech. rep., Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[24] DIJKSTRA, E. W. The humble programmer. *Communications of the ACM 15*, 10 (1972), 859–866.

[25] DIJKSTRA, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*, 8 (1975), 453–457.

[26] DUTERTRE, B. Yices 2.2. In *Computer Aided Verification* (2014), Springer, pp. 737–744.

[27] EDMUNDS, A., AND BUTLER, M. Tool support for event-b code generation.

[28] FERREIRÓS, J. The road to modern logican interpretation. *Bulletin of Symbolic Logic 7*, 04 (2001), 441–484.

[29] FITTING, M. *First-order logic and automated theorem proving*. Springer, 1996.

[30] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. Extended static checking for Java. In *ACM Sigplan Notices* (2002), vol. 37, ACM, pp. 234–245.

[31] FURIA, C. A., AND MEYER, B. Inferring loop invariants using postconditions. In *Fields of logic and computation*. Springer, 2010, pp. 277–300.

[32] GE, Y., BARRETT, C., AND TINELLI, C. Solving quantified verification conditions using satisfiability modulo theories. In *Automated Deduction–CADE-21*. Springer, 2007, pp. 167–182.

[33] GE, Y., AND DE MOURA, L. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification* (2009), Springer, pp. 306–320.

[34] GEORGES, A., BUYTAERT, D., AND EECKHOUT, L. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices 42*, 10 (2007), 57–76.

[35] GLUCK, P. R., AND HOLZMANN, G. J. Using spin model checking for flight software verification. In *Aerospace Conference Proceedings, 2002. IEEE* (2002), vol. 1, IEEE, pp. 1–105.

[36] GOLDBERG, D. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR) 23*, 1 (1991), 5–48.

[37] GRIES, D. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming 2*, 3 (1982), 207–214.

[38] HEATH, T. L. *The Thirteen books of the Elements of Euclid, Vol. 2 (Books III–IX)*. New York, NY: Dover, 1956.

[39] HERBERT, L., LEINO, K. R. M., AND QUARESMA, J. Using dafny, an automatic program verifier. In *Tools for Practical Software Verification*. Springer, 2012, pp. 156–181.

[40] HOARE, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (1969), 576–580.

[41] HOARE, T. The verifying compiler: A grand challenge for computing research. In *Modular Programming Languages*. Springer, 2003, pp. 25–35.

[42] KOENIG, J., AND LEINO, K. R. M. Getting started with dafny: A guide. *Software Safety and Security: Tools for Analysis and Verification 33* (2012), 152–181.

[43] LECOMTE, T. Safe and reliable metro platform screen doors control/command systems. In *FM 2008: Formal Methods*. Springer, 2008, pp. 430–434.

[44] LECOMTE, T., SERVAT, T., AND POUZANCRE, G. Formal methods in safety-critical railway systems. In *Proc. Brazilian Symposium on Formal Methods: SMBF* (2007).

[45] LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning* (2010), Springer, pp. 348–370.

[46] LEINO, K. R. M. Automating induction with an smt solver. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg, 2012), VMCAI'12, Springer-Verlag, pp. 315–331.

[47] LEINO, K. R. M. Automating theorem proving with smt. In *Proceedings of the 4th International Conference on Interactive Theorem Proving* (Berlin, Heidelberg, 2013), ITP'13, Springer-Verlag, pp. 2–16.

[48] LEINO, K. R. M. Developing verified programs with dafny. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 1488–1490.

[49] LEINO, K. R. M., AND MONAHAN, R. Dafny meets the verification benchmarks challenge. In *Verified Software: Theories, Tools, Experiments*. Springer, 2010, pp. 112–126.

[50] LEINO, K. R. M., MÜLLER, P., AND SMANS, J. Verification of concurrent programs with chalice. In *Foundations of Security Analysis and Design V*. Springer, 2009, pp. 195–222.

[51] LEINO, K. R. M., NELSON, G., AND SAXE, J. B. Esc/java user's manual. *ESC 2000* (2000), 002.

[52] LEVESON, N. G., AND TURNER, C. S. An investigation of the therac-25 accidents. *Computer 26*, 7 (1993), 18–41.

[53] MÉRY, D., AND SINGH, N. K. Automatic code generation from event-b models. In *Proceedings of the second symposium on information and communication technology* (2011), ACM, pp. 179–188.

[54] MICROSOFT RESEARCH. Z3 - guide. `http://rise4fun.com/z3/tutorialcontent/guide/`. Accessed May 29, 2014.

[55] NELSON, S. D., AND PECHEUR, C. Formal verification for a next-generation space shuttle. In *Formal Approaches to Agent-Based Systems*. Springer, 2003, pp. 53–67.

[56] NEW YORK UNIVERSITY. Cvc4's native language. `http://cvc4.cs.nyu.edu/wiki/CVC4's_native_language#Tuple_Types`. Accessed August 27, 2014.

[57] PEARCE, D. J. Whiley v0.3.29 released! `http://whiley.org/2014/09/01/whiley-v0-3-29-released/`. Accessed Sep 05, 2014.

[58] PEARCE, D. J. A calculus for constraint-based flow typing. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs* (2013), ACM, p. 7.

[59] PEARCE, D. J. Sound and complete flow typing with unions, intersections and negations. In *Verification, Model Checking, and Abstract Interpretation* (2013), Springer, pp. 335–354.

[60] PEARCE, D. J. Getting started with whiley.

[61] PEARCE, D. J. The whiley language specification.

[62] PEARCE, D. J., AND GROVES, L. Whiley: a platform for research in software verification. In *Software Language Engineering*. Springer, 2013, pp. 238–248.

[63] PEARCE, D. J., AND GROVES, L. Reflections on verifying software with whiley. In *Formal Techniques for Safety-Critical Systems*. Springer, 2014, pp. 142–159.

[64] PEARCE, D. J., AND NOBLE, J. Implementing a language with flow-sensitive and structural typing on the jvm. *Electronic Notes in Theoretical Computer Science 279*, 1 (2011), 47–59.

[65] QCON. Tony hoare. `http://qconlondon.com/london-2009/speaker/Tony+Hoare`. Accessed June 05, 2014.

[66] RANISE, S., AND TINELLI, C. Satisfiability modulo theories. *Trends and Controversies-IEEE Intelligent Systems Magazine 21*, 6 (2006), 71–81.

[67] REYNOLDS, A., TINELLI, C., GOEL, A., KRSTIĆ, S., DETERS, M., AND BARRETT, C. Quantifier instantiation techniques for finite model finding in smt. In *Automated Deduction–CADE-24*. Springer, 2013, pp. 377–391.

[68] RUSTAN, K., AND LEINO, M. Developing verified programs with. In *Proceedings of the 4th international conference on Verified Software: theories, tools, experiments* (2012), Springer-Verlag, pp. 82–82.

[69] SCHNEIDER, S. *The B-method: An introduction*. Palgrave Oxford, 2001.

[70] SMT-COMP. Smt-exec. `http://smt-exec.org/`. Accessed Sep 18, 2014.

[71] STALLMAN, R. M., ET AL. *Using and porting the GNU compiler collection*. Free Software Foundation, 1999.

[72] STEPHENSON, A. G., MULVILLE, D. R., BAUER, F. H., DUKEMAN, G. A., NORVIG, P., LAPIANA, L., RUTLEDGE, P., FOLTA, D., AND SACKHEIM, R. Mars climate orbiter mishap investigation board phase i report, 44 pp. *NASA, Washington, DC* (1999).

[73] SUTER, P., STEIGER, R., AND KUNCAK, V. Sets with cardinality constraints in satisfiability modulo theories. In *Verification, Model Checking, and Abstract Interpretation* (2011), Springer, pp. 403–418.