Predicting SMT solver performance for software verification

Andrew HEALY

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science

MAYNOOTH UNIVERSITY

Department of Computer Science
Faculty of Science and Engineering

October 2016

Head of Department:
Dr. Adam WINSTANLEY

Supervisors:
Dr. James F. POWER
Dr. Rosemary MONAHAN

Contents

Abstract iv

Acknowledgements v

List of Abbreviations & Acronyms vi

List of Figures vii

List of Tables viii

1 Introduction 1

1.1 Introducing Why3 2

1.1.1 A developer-facing front-end 2

1.1.2 An intermediate logic language 2

1.1.3 PO-discharging back-end 3

1.2 Thesis Statement 3

1.3 Contributions 4

1.4 Organisation of this thesis 5

2 Literature Review 7

2.1 Why3 and Software Verification Systems 7

2.1.1 Measurement and Metrics in Software Verification . . 9

Software Verification Competitions and Benchmark Repositories 9

Proof Engineering 12

2.2 Software Measurement and Metrics 13

2.2.1 Measurement and Machine Learning 15

2.3 Machine Learning 15

2.3.1 Software Verification and Machine Learning 16

2.3.2 Where4, portfolio-solving, and the intersection of all three disciplines 17

2.4 Conclusion 18

3 Where4 System Overview and Data Collection 21

3.1 Selection of tools and programs 22

3.1.1 Selection of Why3 programs 22

3.1.2 Selection of SMT solvers 23

3.2 Independent/Predictor variables 25
3.2.1 Extracting static metrics from Why3 proof obligation formulæ 25
Example: *first_last* lemma 26
3.3 Dependent/Response variables 28
3.3.1 Execution time 28
Accounting for randomness with confidence intervals 28
3.3.2 Prover output 30
Setting a timeout limit for measurement 31
3.4 Summary 33

4 Choosing a Prediction Model 34
4.1 The benefit of portfolio-solving in Why3 34
4.1.1 The relative utility of solver responses 37
4.2 Classification and regression 38
4.2.1 Predicting the single best solver 38
4.2.2 Predicting the best *ranking* of solvers 38
4.2.3 Predicting solver runtime and response separately . . 39
4.2.4 Combining the prediction of solver response and runtime 39
4.2.5 The Cost Function 39
4.3 Choosing the most effective algorithm for rank prediction . 41
4.3.1 The ML algorithms we used for model training . . . 42
ML Algorithm 1: Support Vector Machines 42
ML Algorithm 2: Decision Trees 43
ML Algorithm 3: Random Forests 44
ML Algorithms 4 & 5: Linear and Ridge Regression . 44
ML Algorithm 6: k-Nearest Neighbours Clustering . 44
4.3.2 Experimental Configuration 45
4.4 Ranking strategies 45
4.4.1 Best Ranking 46
4.4.2 Worst Ranking 47
4.4.3 Random Ranking 47
4.4.4 Quantifying Solver Contributions Using Ranking Strategies 48
4.5 Predictor Selection Results 49
4.5.1 EC1: Time 49
4.5.2 EC2: R^2 Score 50
4.5.3 EC3: Normalised Distributed Cumulative Gain . . . 50
4.5.4 EC4: Mean Average Error 52
4.5.5 EC5: Regression Error 53
4.5.6 Properties of multi-output problems 53

4.5.7 The chosen model . . . 54
4.6 Summary . . . 55

5 OCaml Implementation **56**
5.1 Finding the minimal number of trees . . . 58
5.2 Encoding the random forest . . . 58
5.3 Extracting features . . . 61
5.4 Integration with Why3 . . . 61
5.5 Summary . . . 62

6 Evaluating Where4 on Test Data **63**
6.1 EQ1: How does Where4 perform in comparison to the eight SMT solvers? . . . 64
6.1.1 Use of a cost threshold . . . 67
6.2 EQ2: How does Where4 perform in comparison to the three theoretical strategies? . . . 70
6.3 EQ3: What is the time overhead of using Where4 to prove Why3 goals? . . . 72
6.4 Threats to Validity . . . 73
6.4.1 Internal . . . 73
6.4.2 External . . . 74
6.5 Discussion . . . 75

7 Conclusion **76**
7.1 Future Work . . . 76

A Ocaml Interfaces **78**

B Where4 command-line options **81**
B.1 The Where4 command given to Why3 . . . 82

C Where4 installation options **83**

Bibliography **85**

MAYNOOTH UNIVERSITY

Abstract

Faculty of Science and Engineering
Department of Computer Science

Master of Science

Predicting SMT solver performance for software verification

by Andrew HEALY

The approach Why3 takes to interfacing with a wide variety of interactive and automatic theorem provers works well: it is designed to overcome limitations on what can be proved by a system which relies on a single tightly-integrated solver. In common with other systems, however, the degree to which proof obligations (or "goals") are proved depends as much on the SMT solver as the properties of the goal itself. In this work, we present a method to use syntactic analysis to characterise goals and predict the most appropriate solver via machine-learning techniques.

Combining solvers in this way - a *portfolio-solving* approach - maximises the number of goals which can be proved. The driver-based architecture of Why3 presents a unique opportunity to use a portfolio of SMT solvers for software verification. The intelligent scheduling of solvers minimises the time it takes to prove these goals by avoiding solvers which return *Timeout* and *Unknown* responses. We assess the suitability of a number of machine-learning algorithms for this scheduling task.

The performance of our tool Where4 is evaluated on a dataset of proof obligations. We compare Where4 to a range of SMT solvers and theoretical scheduling strategies. We find that Where4 can out-perform individual solvers by proving a greater number of goals in a shorter average time. Furthermore, Where4 can integrate into a Why3 user's normal workflow - simplifying and automating the non-expert use of SMT solvers for software verification.

Acknowledgements

The first people to acknowledge, even if Umberto Eco finds it distasteful, are my supervisors: Dr Rosemary Monahan and Dr James F. Power. Their guidance and expertise were critical to the development of this project. I thank them for giving me the opportunity to conduct this research.

Secondly, I must thank the broader staff and postgraduate students of the Department of Computer Science. They have made the past two years very enjoyable ones. Special thanks to members of the POP group, past or present, who have been a valuable source of encouragement and domain-specific knowledge. Marie deserves a special mention for proof-reading this thesis.

It is conventional for developers of portfolio solvers to acknowledge the development effort associated with their constituent tools, and I must do the same. In particular, I am indebted to the Why3 system, on which this work relies.

Lastly but most importantly, I thank my parents for all their support.

List of Abbreviations & Acronyms

AI	**A**rtificial **I**ntelligence
ATP	**A**utomatic **T**heorem **P**rover
DSL	**D**omain-**S**pecific **L**anguage
IDE	**I**ntegrated **D**evelopment **E**nvironment
ITP	**I**nteractive **T**heorem **P**rover
IVL	**I**ntermediate **V**erification **L**anguage
ML	**M**achine **L**earning
PO	**P**roof **O**bligation
SAT	**Sat**isfiability
SE	**S**oftware **E**ngineering
SMT(-LIB)	**S**atisfiability **M**odulo **T**heories (-**Lib**rary)
SV	**S**oftware **V**erification
SVM	**S**upport **V**ector **M**achine
TPTP	**T**housands of **P**roblems for **T**heorem **P**rovers

N.B. In the text, to distinguish references to Machine Learning (ML) from references to the `ML` programming language, the latter is typeset in `monospace` font.

List of Figures

1.1 Overview of the Where4 project and this thesis's structure . 5

2.1 Where4 is placed at the intersection of three disciplines . . . 8

3.1 Diagram illustrating the process to collect predictor and response variables for the Where4 model 22
3.2 Tree illustrating the Why syntactic features counted by traversing the AST. 25
3.3 *WhyML* code for the *first_last* lemma in *edit_distance.mlw* . . 27
3.4 The parse tree extracted from the *first_last* goal (Fig. 3.3) . . . 27
3.5 Accounting for errors in measurement of solver execution time x 30
3.6 The relative amount of Valid / Unknown / Timeout / Failure answers from the eight SMT solvers 31
3.7 Cumulative number of useful responses for each solver over 60 seconds 32

4.1 Overview of the process used to derive the Where4 prediction model 42
4.2 Relative solver portfolio contributions 49

5.1 Data design for JSON encoding 59
5.2 Where4 modules 60

6.1 The relative amount of Valid/Unknown/Timeout/Failure answers from the eight SMT solvers and the top-ranking solver from the three theoretical strategies and Where4 (w/(o) presolving). 65
6.2 Determining a cost threshold 68
6.3 The effect of using a cost threshold 69
6.4 The time taken by each theoretical strategy and Where4 to return all *Valid/Invalid* answers in the test dataset of 263 goals 71

List of Tables

2.1 Summary of SV benchmark sources 10
2.2 Summary of ML algorithms used in SV tools 19

3.1 SMT solvers supported by Where4 24
3.2 Non-zero metrics calculated for *first_last* 28

4.1 Results of running eight solvers on the example Why3 programs 35
4.2 Breakdown of results in terms of triviality and hardness . . . 35
4.3 Result of eight solver executions on *first_last* 41
4.4 Predictor Selection Results 51
4.5 Illustration of various nDCG scores 52
4.6 Relative importance of features for Where4 54

5.1 Finding the minimal number of trees 58

6.1 Results for eight solvers, Where4 and three strategies on test set 64
6.2 The effect of using a cost threshold 71

Chapter 1

Introduction

As software development projects grow in size and budget, so does the cost, time, and complexity associated with fixing the problems that inevitably arise. As an example of a particularly expensive error, in 1996 the Ariane 5 rocket exploded soon after launch due to a numerical conversion error: costing $350 million [40]. Software engineering (SE) as a field has introduced concepts, models and techniques intended to combat such problems:

- Software process models (e.g. agile development).
- Specifications for the abstraction of systems (e.g. the Unified Modelling Language [99]).
- Methodologies to ensure the safety of a system (e.g. unit testing).

This thesis is based on tool support for software verification (SV). In terms of the categories listed above, SV is concerned with the safety of a system. In contrast to software testing which ensures a program behaves as *expected* for a given set of inputs, SV ensures the programs behave as specified for *all possible* inputs. SV tools achieve this through logical reasoning based on formal specifications. The consequences for the safety and reliability of complex software systems can be impressive. One SV tool, Atelier B, was used to verify the correctness of software operating the (driver-less) Line 14 of the Paris métro [33].

The three examples of software engineering techniques above have benefited from excellent tool support and have been widely adopted by software developers. Software verification systems, however, often require specialised knowledge and suffer from a lack of integrated tool support [3]. The automation of processes requiring domain-specific knowledge can be an important technique to encourage the adoption of new software engineering practices. This thesis presents a tool to automate one such process for the Why3 [29] software verification system. By providing a layer of abstraction to the Why3 back-end architecture, we hope to make the system more approachable for software engineers without *specific knowledge* of the tools interfaced by the Why3 system.

1.1 Introducing Why3

In general, software verification systems can be viewed as being composed of the following components:

1. A developer-facing front-end.
2. An intermediate logic language (IVL).
3. A back-end to solve proof obligations (POs).

This section will discuss Why3's modular and extensible approach to these three components compared to other SV systems.

1.1.1 A developer-facing front-end

Usually this component takes the form of either a special-purpose programming language (such as Dafny [82]) or the modification of an existing language such as Java (the Java Modelling Language (JML) [38]) or C# (via Spec# [11]). In either case, the language must support verification constructs such as data invariants, pre- and post- conditions.

The WhyML [25] programming language is based on Standard ML. Polymorphic types, a module system and a large library of verified data structures make the language approachable and familiar to software engineers. WhyML programs are natively supported by the Why3 integrated development environment (IDE).

There are a number of projects building alternative front-ends for the Why3 system. The Jessie plugin [87] provides deductive verification capabilities for C programs and the SPARK 2014 [81] environment supports the verification of Ada programs through the Why3 system. Both these projects translate POs directly into the Why IVL.

1.1.2 An intermediate logic language

Assertions about a program's properties are formally translated to a lower-level logic language. The IVL typically supports axiomatisation of the program's constraints and other common concepts such as linear arithmetic, arrays etc. A number of proof obligations (POs) are generated by the translation process. The POs must be satisfied in order for the program to be verified. The Boogie language [12] is the IVL used by both the Dafny and Spec# verification systems.

The logic of Why3's IVL is an extension of first-order logic which supports polymorphic types, algebraic data types, quantifiers, recursive and inductive predicate definitions. As it is the front-end to a wide range of automatic and interactive theorem proving tools, the Why IVL is designed to

be expressive and efficient. The major benefit to using Why3's IVL is the capability to send the generated POs to a number of theorem-proving tools at the back-end. This capability, and the flexibility of the language's logic, has led to a growing number of projects translating the output from Atelier-B [91, 71] and Boogie [7] to the Why logic language.

1.1.3 PO-discharging back-end

In order for each PO to be proven satisfiable or otherwise, they must be sent to a special-purpose program which implements the appropriate decision procedures for the logical theories referenced by the IVL. This back-end component may be a Satisfiability (SAT) solver; in which case, all constraints are translated into a boolean formula of propositional logic with a number of unknown values. The solver's task is to prove whether there is some assignment of these variables which satisfies the formula.

More often, however, a Satisfiability Modulo Theories (SMT) solver is used for SV system back-ends. SMT solvers extend SAT solvers by implementing decision procedures for a number of logical theories. A specific algorithm has been developed to decide problems using arrays or linear arithmetic, for example. The combination of these algorithms allow SMT solvers to prove formulæ using a more expressive range of logical theories than propositional logic. Z3 [47] is an example of a popular and powerful SMT solver. It is used to discharge POs generated by the Boogie, Spec# and Dafny tools.

This component is where Why3 differs most from other SV systems. It implements a modular architecture based on drivers. Using a specific driver file, Why3 writes the input format for each supported automatic theorem prover (ATP), SMT solver, and interactive theorem prover (ITP). Supported ATPs such as MetiTarski [2] and Gappa [50] specialise in solving numerical problems. The supported SMT solvers will be introduced in a later section as their comparison constitutes a main contribution of this thesis. The ITPs supported by Why include Coq [17] and Isabelle [93] (these tools sometimes referred to as *proof assistants*). Each tool's driver file lists the transformations that must take place (e.g. remove polymorphism or inductive predicates) in order for Why3's logic to conform to that of the theorem proving tool.

1.2 Thesis Statement

Having such a range of ATP, SMT and ITP tools supported by a single IVL is a welcome development. Increased operability in the SV domain lets users utilise the most appropriate tool for their specific task. However, time

can be wasted if the wrong tool is consistently chosen. For example, Z3 has a unique and effective approach to reasoning about quantifiers, while the Alt-Ergo [45] SMT solver produces excellent results for POs containing polymorphic types.

By choosing the most appropriate SMT solvers, more POs can be proven and safer software can be developed. Without knowing which SMT solver *is* the most appropriate, however, this choice is essentially a random one. We present a method to automate the choice of SMT solver for any given proof obligation. The resultant tool, which we have named Where4, is a "portfolio solver": it is a portfolio of solving algorithms consisting of several SMT solvers. It attempts to choose the most appropriate of these solvers based on static metrics derived from each PO's syntactic features. Our project's thesis statement:

> *The use of machine leaning and portfolio-solving techniques results in the allocation of SMT resources which can prove more software verification proof conditions than any single solver.*

This work is intended for use by the non-expert developer performing SV as part of an enlightened software development process. This user has no specific knowledge of the internal implementation of individual SMT solvers nor of their relative strengths and weaknesses. They may not even know the particular characteristics of the PO they are trying to prove (hidden as they may be through the use of a front-end specification language). A PO may be characterised by its use of universal quantifiers or non-linear arithmetic, for example.

We limit this work to the use of SMT solvers in order to make valid comparisons about each tool's suitability for SV-specific tasks. As we shall discuss in Sec. 2.1.1, the lack of a standard input format for SV systems makes their comparative evaluation difficult. The opportunity to make such an evaluation is a secondary aim for this project.

1.3 Contributions

The main contributions of this work are:

1. The design and implementation of a portfolio solver, Where4, which uses supervised machine learning to predict the best solver to use based on metrics collected from goals.
2. The integration of Where4 into the user's existing Why3 work-flow by imitating the behaviour of an orthodox SMT solver.
3. A set of metrics to characterise Why3 goal formulæ.

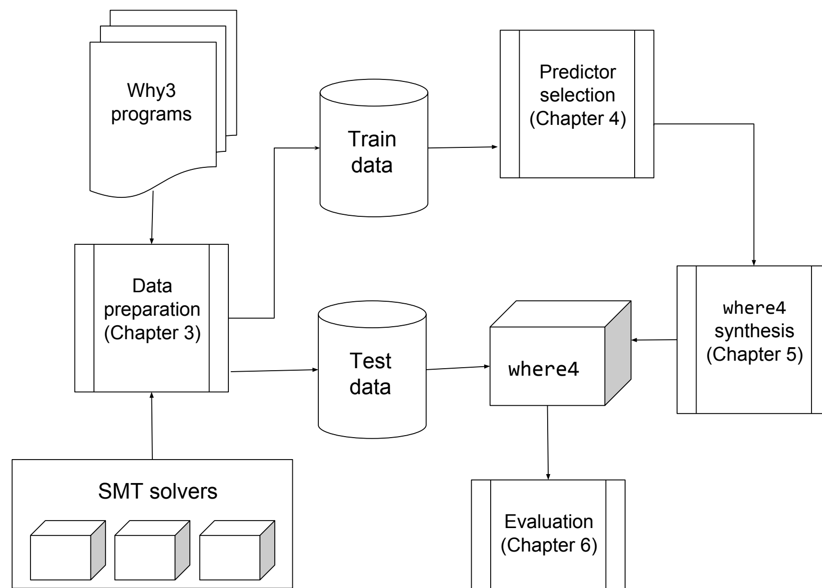

FIGURE 1.1: Overview of the Where4 project and this thesis's structure

4. Statistics on the performance of eight SMT solvers using a dataset of 1048 Why3 goals.

5. An evaluation of six machine learning algorithms' performance when predicting the most appropriate SMT solvers, given a Why3 PO.

1.4 Organisation of this thesis

We view this research as being situated at the intersection of three broad research areas within computer science: software verification, the measurement of software through metrics, and machine learning (ML). Chapter 2's literature review reflects these themes and focusses particularly on their intersection.

The organisation of the rest of this thesis, and the associated data / artefacts produced during the development of Where4, is illustrated in Fig. 1.1. Chapter 3 discusses the choices we made regarding the experimental setup of our empirical study. These choices include the dataset of Why3 POs and the portfolio's individual SMT solvers. The process to choose and extract independent and dependent variables for the machine learning task is also described in this chapter. Chapter 4 contains a discussion of the chosen solvers' performance on the dataset. A number of options considered for the prediction task and an evaluation of several ML algorithms' suitability for this task is contained in this chapter.

Chapter 5 describes our implementation of the chosen model using Why3's OCaml API. The encoding of Where4's ML model into OCaml data structures is discussed in this chapter. Chapter 6 defines three Evaluation Questions designed to assess the final Where4 tool's performance, usability and practicality. We evaluate Where4 with reference to these questions. Chapter 7 concludes this thesis by reflecting on the contributions of our work and suggesting some future directions for this project.

Chapter 2

Literature Review

This chapter views Where4 as incorporating ideas from three separate disciplines: software verification, machine learning, and software measurement and metrics. A Venn diagram illustrating the intersections of these disciplines and how this chapter is organised according to these intersections is given in Fig. 2.1. We shall concentrate this review on Software Verification and SV research incorporating concepts from the other two disciplines. Where more background in associated topics outside the SV domain is required, the literature is discussed in the relevant sections (i.e. Sec. 2.2 and 2.3).

The rest of this chapter will review the literature associated with each segment of Figure 2.1 – moving clockwise from the top. This review was approached with the following questions:

Q1 What does the SV tool landscape look like in terms of interoperability?

Q2 Does a standardised suite of benchmark programs exist for the comparison of deductive SV tools?

Q3 How have concepts from traditional SE been applied to large-scale SV projects?

Q4 How have machine learning concepts been integrated for use in the SV domain?

The answers to these questions have informed the design choices made in regard to the Where4 portfolio-solving tool.

2.1 Why3 and Software Verification Systems

An overview of the Why3 verification system [29, 58] has been given in the previous chapter. The WhyML programming language provides a high-level `ML`-like language for the specification of programs with pre- and post-conditions, recursive definitions and type invariants. An extensive library of verified polymorphic data-types make WhyML a flexible language [26, 25]. It is Why3's driver-based approach to interfacing with external tools

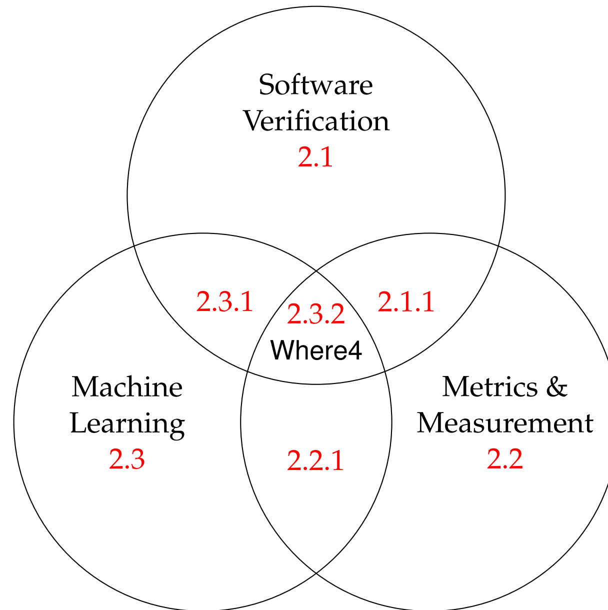

FIGURE 2.1: Where4 is placed at the intersection of three disciplines

that is of most interest to our project. The Why3 approach is, in this regard, quite different from the typical SV system of tightly-integrated systems consisting of an IDE/annotation language/front-end DSL, intermediate logic language, and SMT-solving back-end. Examples of systems which follow this latter model are Spec# [11] and Dafny [82] which use the Boogie [12] IVL and the Z3 [47] SMT solver.

The diversity of languages and formalisms has been matched by an increase in software verification *tools* in recent years. Filliâtre's overview of the deductive verification tool landscape [57] counts sixty-five tools cited or used in the five papers of a special edition of the *Verified Software: Theory, Tools and Experiments* post-proceedings in 2009. Recently, there has been an effort to increase the interoperability of these tools. The two-dozen ATPs, ITPs and SMT solvers targeted by Why3 make it an attractive choice for translations: recent projects have used Why3 as a platform for discharging POs translated from Boogie [7] and the B system [71, 91].

Q1: What does the SV tool landscape look like in terms of interoperability?

For users of SV tools, Why3 facilitates the integrated use of numerous ATPs, SMT solvers and ITPs. This distinguishing characteristic of the Why3 platform makes it arguably the most open platform for the formal verification of software. We chose the Why3 system for the flexibility of its specification language, extensible driver-based architecture and potential for conducting a comparative evaluation of several theorem-proving tools.

2.1.1 Measurement and Metrics in Software Verification

With the diversity of languages, formats, and approaches currently in use in the SV domain, experimental software measurement concepts and techniques must be introduced for the rigorous evaluation, comparison, and characterisation of software. These techniques are usually employed in the general software engineering domain and have been adapted to reflect the specific concerns of formal methods.

Benchmark repositories and tool competitions have proven to be important as a means of evaluation and improvement among SV communities. A number of large-scale SV projects, meanwhile, have necessitated the adaptation of established SE metrics and methodologies.

Software Verification Competitions and Benchmark Repositories

At present, competitions provide the most prominent means of comparing systems which focus on the verification of object-oriented software. In competitions such as those held at FoVeOOS 2011 [32] and the VerifyThis series [69], teams are given the natural-language specifications and pseudocode for a small number of typical SV problems. Any system can be used, with tool developers often choosing to compete using their own system. As well as evaluating solutions based on correctness and completeness, an emphasis is placed on judging a team's implementation *approach* and *ideas*, given the diverse capabilities of the SV systems in use. Previous challenges have been based on a set of eight incremental benchmarks for software verification tools proposed by Weide et al. at VSTTE 2008 [109]. The Why3 development team are regular competitors at VerifyThis and some of the standard Why3 examples are refined versions of solutions submitted at the competition [26].

SV-COMP [18, 19] is a well-established annual competition for automatic program verifiers. The tools use static analysis and model-checking techniques to ensure properties such as reachability or termination. Teams can choose to compete in a subset of categories based on the strengths of their tool. As soundness of automatic program verifiers cannot be guaranteed, marks are deducted for false positive and false negative answers. The scoring hierarchy rewards true positive answers with the highest marks and penalises false positive answers severely; with "Unknown" answers having zero effect on a tool's score. We followed a similar scoring hierarchy when devising Where4's cost function (Sec. 4.2.5).

Importantly, a large, publicly-available benchmark repository consisting of 6661 programs has been developed using these competition questions. It is possible to maintain such a repository due to the standard input format of the tools which all accept input in the C language. Other efforts to

TABLE 2.1: Summary of SV benchmark sources

Benchmark Source	Target Domain	Example tools	Input format	Size	Website (last visited 28/9/16)	Reference
VerifyThis	Program Verification Systems	Why3, Dafny, Spec#, VeriFast, KIV	Natural Language and Pseudo Code	33	verifythis.org	[69]
VSTTE Incremental Benchmarks	Program Verification Systems	Why3, Dafny, RESOLVE	Natural Language	8	-	[109]
VACID-0	Program Verification Systems	Why3, Dafny, KeY, VCC, Coq	Natural Language and Pseudo Code	5	-	[83]
SMT-LIB	SMT solvers	Z3, CVC4, Yices2, veriT, Alt-Ergo	SMT-LIB v2	> 100,000 POs	smtlib.org	[13]
TPTP	ATPs and SMT solvers	Why3, Eprover, Beagle, CVC3, CVC4	TPTP	20,306 POs	tptp.org	[104]
BWARE	ATPs and SMT solvers	Why3, Alt-Ergo, CVC3, CVC4	TPTP / SMT-LIB v2	12,876 POs	bware.lri.fr	[48]
SV-COMP	ATPs and model-checkers	AProVE, BLAST, CPAchecker	C	6661 files	sv-comp.sosy-lab.org	[19]
TIP	ATPs supporting induction	HipSpec, ACL2, Zeno	TIP extended SMT-LIB	343	github.com/tip-org	[42]
The Why3 examples	Program verification through Why3	Why3 and its supported back-ends	WhyML	128 programs, 1048 POs	toccata.lri.fr	[26, 107]

standardise benchmark repositories for software verification tools encountered during this review are summarised in Table 2.1.

The need for a standard set of benchmarks for the diverse range of verification systems was identified by a number of participants in the week-long seminar at Dagstuhl [21] in 2014. The series of workshops and events brought the model-checking and SV system communities together. Qualitative, repeatable comparative evaluation was agreed as an important goal if deductive software verification is to advance as an engineering discipline. Following on from the eight incremental benchmarks proposed at VSTTE [109], the VACID-0 [83] project is another attempt to maintain a repository of standard abstract specifications for verified data structures and operations similar to those used in the VerifyThis competition. The VACID-0 benchmarks also include a marking scheme to identify the most important aspects of the specification for a tool to be able to verify.

In addition to SV-COMP, the benefits of a large benchmark suite written in a common input language are evidenced by the SMT-LIB [14] project. The performance of SMT solvers has significantly improved in recent years due in part to the standardisation of an input language and the use of standard benchmark programs in competitions [44]. In contrast to the ATPs competing in SV-COMP, SMT solvers are assumed to be correct. Therefore, solvers are marked according to how many problems they can solve and the time taken to solve problems. Why3 includes an SMT-LIB printer and uses the format for a number of SMT solvers (including CVC4, Z3, and veriT). Our previous work [62] has exploited this feature: verification tasks were compared to other application domains of SMT solvers using the SMT-LIB repository as a data source.

The TPTP (Thousands of Problems for Theorem Provers) project [106] is a benchmark repository with similar aims to SMT-LIB but has a wider scope. The problems target theorem provers which specialise in numerical problems as well as general-purpose SAT and SMT solvers. The TPTP library is specifically designed for the rigorous experimental comparison of solvers [105]. There has been a significant development effort by Why3 developers to support the TPTP format in Why3 and to extend the TPTP language with rank-1 polymorphism [23] thereby allowing the use of `ML`-like polymorphic types to be used in an interchange format understood by over two-dozen provers. This extension of the TPTP language makes it more similar to the specification language of Why3.

More recently, a group from Chalmers University of Technology has introduced a repository of benchmarks for inductive theorem provers called Tons of Inductive Problems (TIP) [42]. The project aims to facilitate the comparison of how various tools handle proving inductive properties and uses

a specialised problem description language which is an extension of SMT-LIB. An interesting aspect of the project is its approach to interoperability: aside from benchmarks written in the aforementioned format, tools are provided to translate the problems into standard SMT-LIB, Haskell, TPTP, Isabelle/HOL and WhyML.

Q2: Does a standardised suite of benchmark programs exist for the comparison of deductive SV tools?
Although the need for such a benchmark suite is well understood, the diversity of input languages hampers progress towards this goal. A small number of abstract specifications for standard problems, mostly collected by competitions which evaluate *approaches* used by various systems to software verification, are the closest approximation.

Proof Engineering

The scale of formal software engineering projects has grown in recent years. The formal verification of the seL4 microkernel [79] and Thomas Hales' *Fly-Speck* proof of the Kepler Conjecture [60] are large and complicated engineering projects developed over a number of years. Both projects represent significant engineering efforts – it is estimated that the seL4 verification took twenty-five person-years of work – and produced a large volume of software artefacts in the form of *proof scripts*. Researchers applying concepts from software engineering to manage and measure such projects call their practice "proof engineering" [78]. Proof engineering has become an active research area in recent years.

Both object-oriented software and formal proofs make use of "modules" to package related classes and lemmas/axioms respectively. Aspinall and Kaliszyk [8] suggest that this common approach to modularity allows the standard Chidamber and Kemerer [41] (CK) metrics to be adapted for formal SE projects. The authors use this analogy to model and measure the dependency tree for proof modules and the derivation of the module's coupling and cohesion metrics (CK metrics will be discussed further in Section 2.2).

Other approaches measure syntactic features of the specification to derive complexity metrics. The verification of the seL4 microkernel mentioned previously was used as the basis for at least two similar studies. For this large-scale SV project, the property statement *size* was found to be quadratically related to the human effort (and associated cost) of development [88]. Staples et al. argue that code sizing (i.e estimating the number of lines of executable C code that will need to be written) is more strongly correlated to the size of the formal specification (both abstract and executable) than to a metric based on a notion of "function points" [102]. The method of sizing software using function points is an international standard based

on the identification of discrete processes described by a software requirements document [1].

The *quality* of specifications for a single project (the Web-Service Definition Language) was observed over a period of time in a study by Bollin [30]. Formal specifications, written in the Z language [112], were measured for their cohesion and coupling. The measurement of specifications is in contrast to the previous examples [8, 88] which used the proof scripts.

The proof obligation formulæ used by Where4 are more similar to specifications than to the proof scripts used in interactive theorem proving. Formal specifications give no indication of *how* the formula is to be proved, only providing a precise description of properties which *must* hold. Procedural proof scripts (such as the Isabelle style of theorem proving) can be thought of as similar to conventional programs because they combine formal specifications and tactic applications to produce machine-verifiable proofs.

Formal specification is facilitated by the Object Constraint Language (OCL) [43] in UML models. Two studies propose complexity metrics for OCL expressions. The first [98] uses structural metrics such as the number of operators, quantifiers, etc. A later study [39], however, takes the view that dynamically measuring the *number of objects* involved in the expressions evaluation provides a more accurate measure of complexity. As this second approach is more specific to object-oriented software, we chose to use mostly structural metrics as predictor variables for Where4.

Q3: How have concepts from traditional SE been applied to large-scale SV projects?

The application of SE concepts to large-scale SV projects provides an emerging set of metrics by which we can characterise formal software artefacts. Structural (*internal*) metrics have been defined in order to predict *external* measures such as effort and cost estimation. McCabe's complexity metric and the CK suite have been adapted for use in the SV domain. As will be discussed later in this thesis, the appropriate selection of predictor variables is important for all machine learning tasks. Traditional SE metrics have been used as a solid basis for predictor variables in projects which combine ML techniques with formal verification.

2.2 Software Measurement and Metrics

The selection of independent variables used by Where4 was guided by the research discussed in the previous section. This section gives context to this research by situating the need for software metrics in the wider SE domain.

Just as software verification aims to make software safer and more dependable by using formal methods to prove its correctness, software metrics can be used to quantify the characteristics of a program which make it more likely to fail. The rigorous measurement of software and the definition of metrics to group and comprehend these measurements is vital for the accurate prediction of a project's development schedule, cost, associated lines of code, etc. A comprehensive overview of the topic is given in Fenton and Pfleeger's book [56]. The metrics of most interest to our project are *internal, structural* code-based metrics. Those introduced in Sec. 3.2 are of this type.

We have previously made reference to CK metrics in this chapter. The CK metric suite was developed in response to the popularity of object-oriented SE practices. Weighted Methods per Class, Depth of Inheritance Tree and Coupling Between Object classes are examples of some CK metrics. As an example of its use, the suite has been relatively successful for prediction of maintenance effort [84]. As our POs are taken as simple, stand-alone formulæ, (rather than being organised in classes or modules) we do not use CK metrics. This is in contrast to the projects using interactive proof scripts discussed in the previous subsection.

One particularly useful metric for measuring code complexity was defined by McCabe in 1976 [89]. The graph-based cyclomatic complexity of a function is a size-independent and intuitive measure of the code's complexity. To calculate the McCabe cyclomatic complexity of a piece of code, a control flow graph representation is constructed where each indivisible sequence of statements is a node. Decision nodes are typically constructed from `if/else` statements. The number of nodes is subtracted from the number of lines connecting those nodes plus 2. For low-level languages, McCabe showed that the cyclomatic complexity of code with a single entry and exit point is equal to the number of decision nodes [89]. It has proved useful in unit-testing scenarios [90]. It is also used to estimate *external* metrics such as how long a project is expected to take, or how much it will cost. McCabe's complexity metric has previously been adapted for use in measuring the complexity of context-free grammars [96].

While structural metrics such as McCabe's are statically measured, the statistically-accurate *dynamic* measurement of software is important if the predicted behaviour of a program is to be related to the *actual* observed behaviour. Many of the associated issues are addressed by Lilja [85] in his book on the subject. Robust experimental methods for software engineering are the subject of other major studies [100], including recent work by Kitchenham et al. [76] who propose the use of kernel density plots as a visualisation method to gain a better understanding of data distributions for empirical software engineering.

2.2.1 Measurement and Machine Learning

At the intersection of software metrics, measurement, and machine learning, there has been a growing trend of using ML techniques to make predictions about SE metrics. Examples of this trend include the use of text-based clustering to predict defect resolution time [9], effort estimation using Support Vector Machines (SVMs) [101] and the use of genetic algorithms for the efficient allocation of cloud-based resources [68]. In a survey on the topic, Gandotra et al. [59] cite numerous examples of the use of ML classification algorithms to identify potentially dangerous or intrusive software.

Recent issues of the journal *Empirical Software Engineering* contain many more articles combining the use of ML techniques and standard software metrics. Zhang and Tsai [116] edited a collection of such journal papers.

2.3 Machine Learning

For a broad overview of the extensive subject of Machine Learning, we refer the reader to three useful overviews. Mitchell [92] and Bishop [22] both balance practical implementation issues for a variety of algorithms with a treatment of their theoretical foundations. Domingos [51] focuses on new research on comparing and combining ML approaches while also giving an entertaining account of the history of many ML algorithms. We focus this section on some of the background literature and considerations associated with the algorithms compared during the development of Where4 (Sec. 4.3). More details about the individual algorithms will be given in Sec. 4.3.1, but a brief introduction to three algorithms discussed in this chapter – SVMs, K-Nearest Neighbours, and Random Forests – is given in this section. By briefly introducing a number of algorithms at this stage, we hope that the reader is not lost amongst unfamiliar concepts if ML is new to them.

SVMs [46] have been mentioned in the previous subsection. Briefly, the algorithm aims to find a number of hyperplanes or (*support vectors*) in an n-dimensional space (where n is the number of features) so that the distance between any hyperplane and any training instance is maximal. Accurate and robust predictions about the characteristics of a testing instance can be made by asking the algorithm which support vectors define its position.

K-Nearest Neighbours (k-NN) also utilises a distance metric in n-dimensional space for its predictions. Instead of finding separating hyperplane, however, a notion of *similarity* is used to group instances into homogeneous clusters. Where SVMs are robust to outliers, the k-NN algorithm is more sensitive, and care has to be taken with the distance metric used.

After analysing a number of ML approaches (as discussed in detail in Chapter 3) Where4 ultimately uses a Random Forest [35] method for prediction. The Random Forest approach is an ensemble extension of the Decision Tree [97] algorithm. A decision tree is constructed by recursively splitting the set of training instances based on the feature and threshold resulting in the largest information gain. A random forest takes the average of a number of tree's predictions where the individual trees have been constructed using a random subset of the training instances and/or features.

2.3.1 Software Verification and Machine Learning

As this project deals with training ML algorithms on software artefacts, it relies on representations of these programs. As discussed in Sec. 2.2, the measurement of software requires the use of metrics. The intersection of software verification and machine learning, therefore, necessarily involves concepts from the software metrics and measurement domain. This subsection concentrates on research from the field of interactive theorem proving while the next looks at portfolio solvers in more detail.

We have already introduced the Flyspeck [60] proof of the Kepler conjecture. The 14,185 theorems and millions of lemmas form the basis for a number of projects which aim to take advantage of ATP tools within interactive environments [74, 75]. These projects involve the translation of TPTP formulæ from the higher-order logic of the interactive prover HOL Light to the input format of a number of ATPs. Kaliszyk and Urban were also involved with extending the Sledgehammer [94] tool for Isabelle [93] with machine learning capabilities. The resulting MaSh [24] engine uses a Naïve Bayes algorithm and clustering to select facts based on syntactic similarity.

Tactic learning is a method applied to automate the interactive theorem proving process. It works by making a logical association from groups of low-level commands to the syntactic and semantic properties of the problem. The groups of commands, called *tactics*, are then applied when a similar problem is recognised. An early example of this approach [72] uses a custom learning algorithm to learn patterns in the ΩMEGA proof system.

The AI4FM[1] group includes researchers from Edinburgh, Newcastle and Heriot-Watt universities. Much of the group's work [55, 64, 80, 37] continues the *tactic learning* line of research introduced in the previous paragraph. For a number of SV platforms, the group uses ML/AI to automate the interactive theorem proving process by learning proof patterns from human experts and proof repositories. The resultant tools such as ACL2(ml) [64] and ML4PG [80] use clustering to identify proof patterns which can be translated to ATPs in a process called *premise selection*. The goal is to

[1] http://www.ai4fm.org

automate and guide interactive proof sessions, thereby increasing the use of ITP tools in practical scenarios. Hazel Duncan's NewT (New Tactic tool) [52] uses a combination of techniques from probabilistic reasoning, machine learning and genetic programming to this end.

These tactic learning tools differ from Where4's approach by mostly using *unsupervised* learning techniques such as clustering. Random Forests have been used for premise selection [54] outside the AI4FM group, the authors claiming an improvement on the performance of *k-means* clustering. In Maynooth University, analogical reasoning (which is a *supervised* method) has been applied to recognise similar computer programs of the Spec# SV system. The goal of the Aris [61] tool is to increase the reuse of formal specifications by automatically matching specifications to implementations.

2.3.2 Where4, portfolio-solving, and the intersection of all three disciplines

Portfolio solving can have a number of meanings. For example, the Z3 SMT solver can run in "portfolio mode": a number of Z3 instances with different heuristics are run in parallel in order to return a result in a faster time [110]. In the case of Where4 and the other projects discussed in this subsection, however, portfolio solving is taken as being the combination of separate ATP tools, used as "black-box" algorithms, selected based on features of the PO being solved.

Portfolio-solving approaches have been implemented successfully in the SAT domain by SATzilla [114]. Early versions of this tool used a ridge regression method to predict runtime for a number of solvers based on an empirical hardness model derived from problem instances. The current version [115] uses a forest of decision trees. SATzilla has been successful in SAT competitions and several portfolio SAT solvers have been developed in recent years [10]. The proceedings of the 2012 SAT competition list the participation of eight portfolio solvers. The SAT competition has had separate tracks and medals for portfolio solvers since 2012.

A team from University College Cork show that instances of another NP-complete problem, the Constraint Satisfaction Problem (CSP), can be solved by translating them to a SAT instance and using a portfolio of solvers [70]. Their evaluation of several learning algorithms identified linear regression as the best model for their data. Other portfolio solvers have been implemented to solve constraint programming and constraint optimisation problems. One such tool is sunny-cp [6] which uses a trained k-Nearest Neighbours model to match solvers based on program features. Short but useful surveys of portfolio solvers in these domains are provided by Amadini et al [4, 5].

Portfolio solvers have also been implemented in the model-checking SV domain. The MUX [108] solver uses metrics derived from syntactic features as input variables. SVMs are trained using counts of datatypes such as scalars, arrays and pointers as well as meta-features such as McCabe complexity. The industrial dataset of proprietary device drivers on which MUX is trained and tested is not publicly available, however.

The large SV-COMP repository of C programs (mentioned in Sec. 2.1.1) has been used to train and test a similar portfolio solver to MUX. The Verifolio [49] solver uses a different SVM weighting function and an extended suite of metrics for C programs based on data-flow analysis. Verifolio was found to be the hypothetical winner of both the 2014 and 2015 editions of SV-COMP.

Neither of the previous two studies include an evaluation of a range of learning algorithms: they are predicated on the use of SVMs. Consequently, this thesis represents a wider treatment of the various prediction models available for portfolio solving.

Q4: How have machine learning concepts been integrated for use in the SV domain?

The practice of premise selection has seen much research into the use of ML techniques to automate interactive proof sessions. This is due in large part to the extensive corpus of proof scripts available to projects working in the ITP domain. SAT solvers have been innovative in their use of ML to produce successful portfolio solvers. Examples of ML in deductive SV systems and SMT solving are less common. Table 2.2 summarises the ML algorithms used by the literature reviewed in this chapter. With the exception of Naïve Bayes, these are the algorithms we chose to evaluate for use by Where4 in Chapter 4.

2.4 Conclusion

This chapter has presented an overview of the research related to this thesis. The intersection of software verification and machine learning domains, in particular, is an active and fertile research area. It is exciting to see research in software verification incorporate recent advances in machine learning. It is equally important, however, that the new possibilities ML creates for verification tools can be evaluated and measured using established metrics and techniques.

We began this review by discussing how the interoperability features of Why3 make it unique among the range of program verification systems. The number of tools which can be used by Why3 offers opportunities to conduct empirical evaluations of a number of SMT solvers. The study can

TABLE 2.2: Summary of ML algorithms used in SV tools

ALGORITHM	GENERAL USE	BRIEF DESCRIPTION	APPLICATION IN SE+SV
Support Vector Machines (SVM)	Binary Classification. Can be adapted for regression and multi-class uses	Find a number of hyperplanes to separate the data with a maximal margin	Classification and regression of C programs for use by model checkers [108, 49]
Naïve Bayes	Classification problems (spam filtering); can be adapted for regression uses	Probability prediction based on Bayes's rule assuming all features are linearly-independent and a given distribution for errors	Sledgehammer's MaSh engine for premise selection [24]
Decision Trees	Multi-output, multi-label classification & regression problems	Recursively partition data by thresholding features based on maximising information gain	Recent versions of SATzilla [115] use multiple decision trees for prediction of solver 'cost'
Random Forests	Can be used on the same problems as Decision Trees	Multiple Decision Trees which use a subset of features and/or data. Designed to minimise overfitting	Adapted for use in premise selection [54]
Clustering	Multi-output classification	Requires a distance metric for identifying groups of similar data. Can be used in *unsupervised* learning where a ground truth is unknown	Premise selection tools for ITPs: ML4PG [80] and ACL2(ml) [64]. Also used by the MaSh engine [24] and sunny-cp [6]
Multiple linear regression	Single-output regression. Can be adapted for many variations	Estimates the parameters of linear prediction functions	The portfolio SAT solver optimised for CSP instances [70]. The first version of SATzilla [114] used a related technique.

be specifically targeted at the solvers' behaviour when given a software verification workload in a common input format. By employing ideas from the software metrics domain as well as current machine learning techniques, a portfolio of solvers can be implemented for the Why3 platform. The goal of our portfolio solver is to reduce the time needed to prove large numbers of POs. It is similar to the portfolio solvers developed for the SAT domain and static model checking.

The next chapter begins by listing seven questions important to the design of Where4. This chapter has presented the research context in which these questions are answered. Where4 is the first portfolio solver, to the best of the author's knowledge, specifically designed for the use of SMT solvers in software verification.

Chapter 3

Where4 System Overview and Data Collection

As empirical studies require many choices to be made from the outset, we identified the following questions which required consideration during the initial planning of Where4:

1. Which solving back-ends of Why3 should be supported by Where4?
2. What program data should the machine learning algorithm use for training and testing?
3. What are the predictor variables to be extracted from these programs?
4. What is to be predicted by the machine learning algorithm?
5. How is the accuracy of response variables to be ensured?
6. Which machine learning algorithm should be used by Where4?
7. How is Where4's interaction with Why3 implemented?

In this chapter we detail the tools and data we chose to measure and the methods used for this measurement; i.e. questions 1-5 are answered. Question 6 is answered in Chapter 4. The choices made in regard to Question 7 are discussed in Chapter 5.

A diagram illustrating this part of the experimental process is given in Fig. 3.1. Verification POs undergo two processes: (i) static syntactic analysis is used to derive feature vectors, and (ii) the result of proving the PO using several SMT solvers is recorded. For each PO in the dataset, the static feature vector is associated with the dynamic measurements for each SMT solver to form our database. Three quarters of the database will be used in the next chapter for training and validation of the ML models. The same data will be used to train the Where4 tool in Chapter 5. The remaining quarter of the database forms the basis of Chapter 6's evaluation.

In Sec. 3.1.1 the repository of verification programs used for training and testing purposes are introduced. The selection of SMT-solving tools to

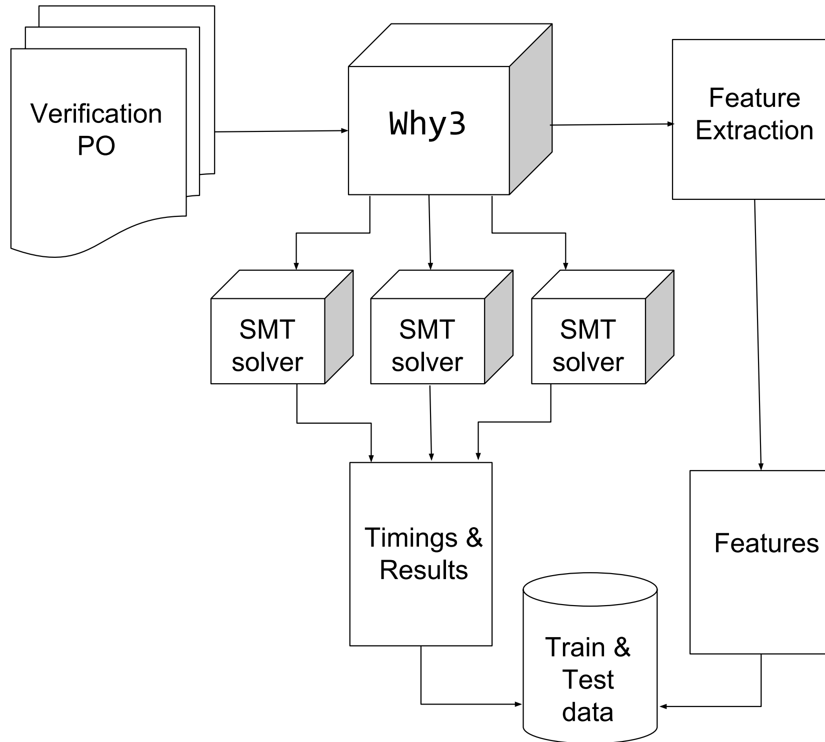

FIGURE 3.1: Diagram illustrating the process to collect predictor and response variables for the Where4 model

be considered by this study is discussed in Sec 3.1.2. Sec. 3.2 details the process taken to extract a number of structural code-based metrics from the proof obligations sent to the SMT solvers by Why3. These metrics are the predictor variables for the Where4 models; otherwise known as *independent* variables. The *response* (or *dependent*) variables are predicted by the model. Execution time and solver output are the two aspects of solver behaviour we need to accurately measure in order to characterise solver performance. The steps taken to ensure the response variables are statistically representative are given in Sec. 3.3.

3.1 Selection of tools and programs

3.1.1 Selection of Why3 programs

As we mentioned in Chapter 2, Why3 was chosen for its modular architecture and ability to read from, and write to, many formats associated with software verification tools. The diversity of input languages in the SV domain was referenced in the previous chapter (Sec. 2.1.1, [21, 57]). We refer the reader to the summary of major benchmark repositories for SV tools given in Table 2.1. Given the lack of a large standard benchmark repository for software verification systems, we chose to make use of the 128 example programs written in the WhyML programming language as our corpus

for training and testing purposes. These programs are distributed with the Why3 software; we used version 0.87.1. Many of the programs are solutions to problems posed at software verification competitions such as VerifyThis [26], VSTTE [77] and COST [32]. Other programs are implementations of benchmarks proposed by the VACID-0 [83] initiative.

It is our assumption that the Why3 examples are representative of software verification workload. Two alternatives to this dataset are the TPTP [106] library and the BWARE [48] collection of industrial proof obligations. The latter dataset consists of Atelier-B POs translated into the TPTP format [91]. The TPTP repository is limited to first-order logic problems: inductive problems cannot be expressed. The BWARE SMT-LIB translation uses an upper-bound logic of UFNIA – this set of theories does not use arrays or real arithmetic[1]. Both datasets are, however, much larger. The TPTP library currently consists of 20,306 POs [103] – many of which are very similar to each other. There are 12,876 BWARE benchmarks. In comparison, the 128 WhyML programs produced 1048 individual proof obligations.

The TIP (Tons of Inductive Problems) benchmarks [42], while interesting, were not considered for use as they only measure one aspect of a set of SV tools: their ability to discharge proof obligations which require the use of induction.

Importantly, the chosen dataset makes use of the full capabilities of the Why3 system: the programs include inductive problems and require the use of as many logical theories as each individual SMT solver can reason with.

3.1.2 Selection of SMT solvers

We used six current, general-purpose SMT solvers supported by Why3:

- **Alt-Ergo** is a general-purpose SMT solver written in OCaml (the others in this list are written in C++). It is the most tightly-integrated solver supported by Why3: it supports polymorphic types and its native input format is a previous version of the Why3 intermediate language. Two recent major versions, 0.95.1 and 1.01, are used by Where4.

- **CVC3** is the open-source SMT solver developed by the University of Iowa and New York University. We used version 2.4.1.

- **CVC4** is an entirely re-implemented update to CVC3. Version 1.4 is used by Where4.

[1]More information about the SMT-LIB standard for logical theories can be found at http://smtlib.cs.uiowa.edu/logics.shtml

TABLE 3.1: SMT solvers supported by Where4

SOLVER	LICENCE	WHY3 DRIVER OUTPUT	REFERENCE
Alt-Ergo	CeCILL-C	`.why`: Alt-Ergo native input format	[45]
CVC3	Open-source	`.cvc`: CVC3 native input format	[15]
CVC4	BSD	`.smt`: SMT-LIB version 2	[16]
veriT	BSD	`.smt`: SMT-LIB version 2	[34]
Yices	Non-commercial use	`.ycs`: Yices native input format	[53]
Z3	MIT	`.smt`: SMT-LIB version 2	[47]

- **veriT** is an open-source SMT solver developed by the University of Lorraine, France, and the Federal University of Rio Grande do Norte, Brazil. We used the current version, 201506, which is not officially supported by Why3 v.0.87.1 but is the only version available.

- **Yices** is developed by SRI International and is free for non-commercial use. Where4 uses version 1.0.38 rather than Yices2 because the newer implementation does not support quantifiers – making it unsuitable for SV.

- **Z3** is the SMT solver developed at Microsoft Research. The source code has been available since 2012 and it has been open-source since 2015. Where4 uses two versions: 4.3.2 and 4.4.1.

As Table 3.1 shows, the six solvers use four different input formats between them – three of which are specific to the solver in question. This gives some idea of the interoperability capabilities of Why3 and the difficulties of comparing tools with a diverse range of input languages.

Using two versions of Alt-Ergo and Z3 affords the user more flexibility in their local SMT solver installation. We describe the process Where4 uses to find and use supported SMT solvers on a user's local machine in Chapter 5.

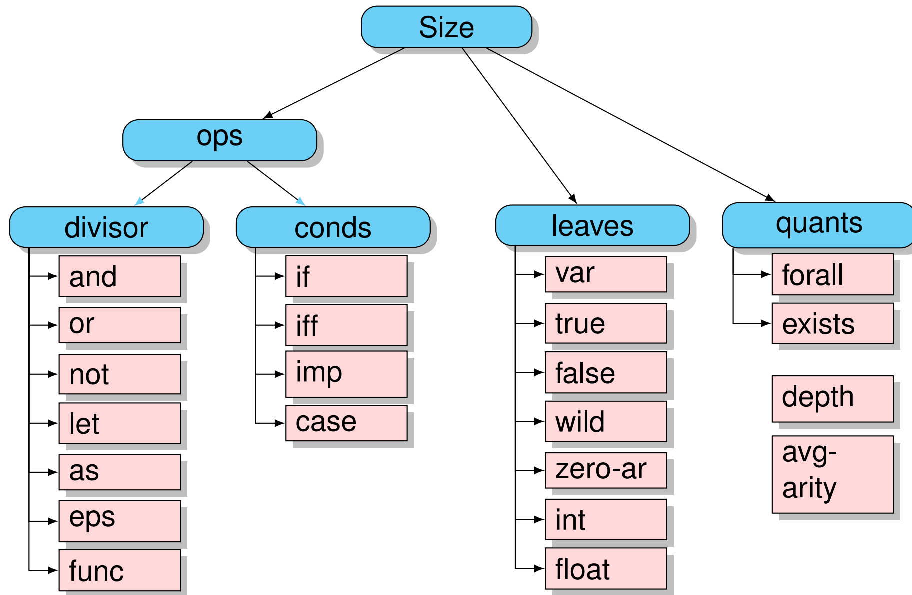

FIGURE 3.2: Tree illustrating the Why syntactic features counted by traversing the AST.

3.2 Independent/Predictor variables

In supervised machine learning terms, these variables represent the known properties of the item from which we want to derive a prediction (in the present case, this item is a computer program). The predictor variables must be an accurate characterisation of the program in order for the ML model to be effective. We chose to use the proof obligations from goals and lemmas rather than those from axioms and predicates (which tend to be repeated in files using the same logical theories). Proof obligations are generated from the goals and lemmas only, while the axioms and predicates provide the context for these POs to be proved by the SMT solver.

3.2.1 Extracting static metrics from Why3 proof obligation formulæ

To extract the structural metrics from the logical formulæ, we traversed the abstract syntax tree (AST) representation. We made use of the Why3 OCaml API to do this. Our approach is similar to the method used internally by Why3 to derive a *shape* string from an interactive proof session [27]. The purpose of shape strings in this context is to track changes in the POs and avoid re-proving files unnecessarily. The shape acts as a minimal fingerprint representing the structure of the PO formula. Instead of producing a string after traversing the AST, our process simply counts the syntactic features of the formula to construct a feature vector.

Fig. 3.2 lists the predictor variables that were used in our study. All of these are (integer-valued) metrics that can be calculated by analysing the

Why3 goal statement, and are similar to the *Syntax* metadata category for proof obligations written in the TPTP format [106]. The metrics we chose to characterise Why3 proof obligations are intended to generalise for first-order formulæ and other IVLs such as Boogie [12] rather than be specific to Why3 syntax. The metrics' simplicity also makes the ML models more transparent and understandable: metrics derived from multiple complicated interactions of structural features would be hard to reason about and apply to other related contexts where learning would be useful.

The features shown in pink rectangles in Fig. 3.2 are counted individually by traversing the AST. The rounded blue nodes represent metrics that are the sum of their children in the tree. The metrics represented by pink rectangles measure syntactic features and are self-explanatory except for the following:

zero-ar: The number of functions which do not take any arguments (i.e. zero-arity functions).

depth: The depth of the AST.

avg-arity: The total number of arguments for all functions which are children of the *divisor* node, divided by the value of *divisor*.

Size measures the size the expression and is the sum of *ops* (the number of operators), *leaves* (the number of leaf nodes in the AST), and *quants* (the number of quantifiers). We map our notion of *conds* to the number of decision nodes used to calculate McCabe's cyclomatic complexity metric discussed in Sec. 2.2.

Options for the extraction of features from WhyML programs were limited. The domain specific language makes this process more difficult in comparison to related work which uses the general-purpose C language [49, 108]. As a result, we decided to use the purely syntactic analysis outlined in this subsection (along with meta information such as the size of the expression, average arity and depth of AST). Keeping the choice of independent variables simple also has the effect of increasing generalisability to other formalisms such as Microsoft's Boogie [12] intermediate language. A direction for future work disscussed in Chapter 7 is to investigate more elaborate methods for feature extraction.

Example: *first_last* lemma

As a minimal illustrative example, we refer to the code listing in Fig. 3.3. This code is one of the 13 POs from the `edit_distance.mlw` file in the Why3 examples directory. The entire program verifies an algorithm which

```
lemma first_last:
   forall a: char, u: word. exists v: word, b: char.
   v ++ Cons b Nil = Cons a u /\ length v = length u
```

FIGURE 3.3: *WhyML* code for the *first_last* lemma in *edit_distance.mlw*

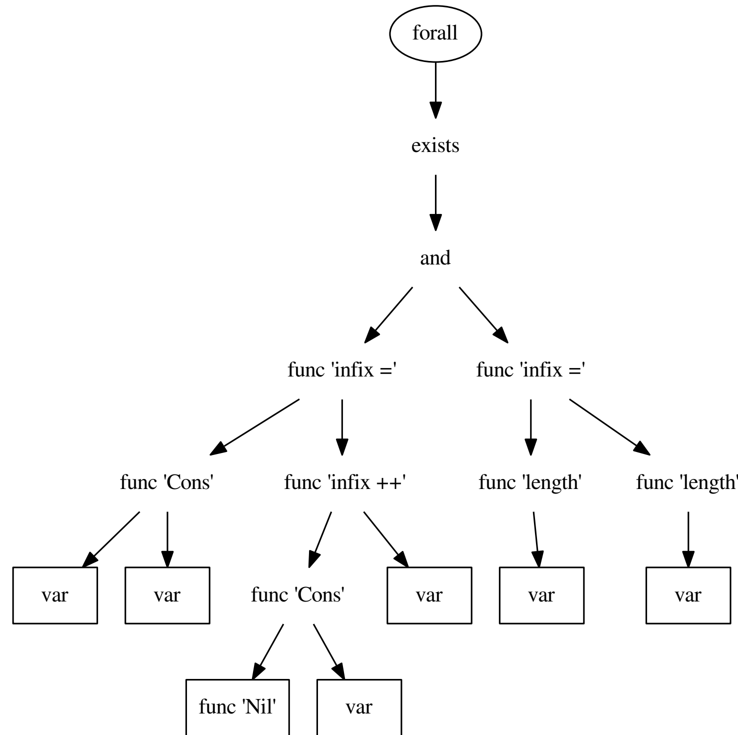

FIGURE 3.4: The parse tree extracted from the *first_last* goal (Fig. 3.3)

finds the "edit distance"[2] similarity measure between two strings. Informally, this intermediate lemma asserts that the same words are produced by (i) prepending a word (minus its first letter *a*) with *a* and (ii) appending a word (minus its last letter *b*) with *b*. A representation of the parse tree from this lemma is shown in Fig. 3.4. The root node (*forall*) is circled and the leaf nodes are shown as rectangles. The name of each *func* (as interpreted by Why3) is shown for clarity. The reader will note that the zero-arity function *Nil* is a leaf node in the tree, while all other functions (*Cons*, =, ++, *length*) and operators (*and*) have an arity of either 1 or 2.

Table 3.2 shows the non-zero metrics used to describe the *first_last* formula for predictive purposes.

[2] https://en.wikipedia.org/wiki/Edit_distance

TABLE 3.2: Non-zero metrics calculated for *first_last*

Metric	Value		Metric	Value		Metric	Value
and	1		exists	1		forall	1
var	6		func	8		quants	2
ops	9		leaves	7		depth	7
size	17		divisor	9		avg_arity	1.55
zero_ar	1						

3.3 Dependent/Response variables

Our evaluation of the performance of a solver depends on two factors: the time taken to calculate the result, and the solver's output as interpreted by Why3.

3.3.1 Execution time

In order to accurately measure the time each solver takes to return an answer, we used a measurement framework specifically designed for use in competitive environments. The BenchExec [18, 20] framework was developed by the organisers of the SV-COMP [19] software verification competition to reliably measure CPU time, wall-clock time and memory usage of software verification tools. We recorded the time spent on CPU by each SMT solver for each proof obligation.

Accounting for randomness with confidence intervals

Any effort to measure execution time accurately is hampered by random errors introduced by the experimental environment. Such errors are inherent to measuring time in real world computing environments due to factors such as cache misses, competing processes and instructions needed to perform the measurement itself. These errors affect the *precision* of the experiment: i.e. how likely a result is to be repeated *exactly* across multiple experiments. To account for these errors, we used the methodology described by Lilja [85] to obtain the number of measurements needed to make an approximation of the true mean execution time.

As each solver execution can be quite expensive in terms of time (a worst case scenario for one measurement of each PO: 1048 (POs) $\times$ eight (solvers) $\times$ ten seconds (time limit) $\approxeq$ 23.3 hours of computation time), a small number of initial measurements (five) were made. From these sample measurements, the mean execution time $\bar{x}$ and standard deviation s were computed.

To determine the number of measurements (n) required for a statistically-significant sample, Lilja recommends that we assume the random errors have a $Student's$ t-distribution (similar in shape – although

more spread out – to a Gaussian distribution). An assumption of the error's distribution is required as there is no *reference value* (ie. the absolutely *correct* time taken by the solver to execute) available. This distribution is chosen as the number of sample measurements taken (< 30) is small whereas if it was large, we could assume errors follow a Gaussian distribution. When comparing the two distributions, differing results are most obvious when we use a very small (< 4) degree of freedom to determine the confidence interval (ie. measurements at the higher and lower end of the range are given more importance). As the following example using seven degrees of freedom shows, this effect is not applicable to our methodology.

We use the five measurements to find the confidence interval (c_1, c_2) using the equations:

$$c_1 = (1 - e)\bar{x} = \bar{x} - z_{1-a/2;n-1}\frac{s}{\sqrt{n}} \tag{3.1}$$

$$c_2 = (1 + e)\bar{x} = \bar{x} + z_{1-a/2;n-1}\frac{s}{\sqrt{n}} \tag{3.2}$$

Either Equation 3.1 or 3.2 can be used to find

$$z_{1-a/2}\frac{s}{\sqrt{n}} = \bar{x}e. \tag{3.3}$$

Solving for n gives

$$n = \left(\frac{z_{1-a/2}s}{e\bar{x}}\right)^2 \tag{3.4}$$

where z is a value from the t-distribution which is used to model the measurement error. An illustration of this distribution, Fig. 3.5, shows that there is a probability $1 - a$ that the actual value being measured (i.e. x: the execution time for each solver to return an answer for a particular PO), is within the confidence interval (c_1, c_2). As the bounds of (c_1, c_2) increase outward, the confidence level increases. We can be 100% confident that the actual mean value is within the interval $(0, \infty)$ but this interval is not practical. A value of 0.1 was chosen for a, meaning we can be 90% confident that the actual value of x is between c_1 and c_2. We allow the computed mean value to be within 7% of the actual mean (i.e. seven degrees of freedom or an allowed error of $\pm$3.5%). Thus for the equations 3.1 to 3.4, we take $e = 0.035$.

Example: finding Alt-Ergo's execution time for *first_last*:
To show how this method is used in practice, we return to the example introduced in Sec. 3.2.1 – the *first_last* goal from **edit_distance.mlw**'s **Word** theory. The five sample measurements of CPU time spent by Alt-Ergo (to return an answer of *Unknown*):

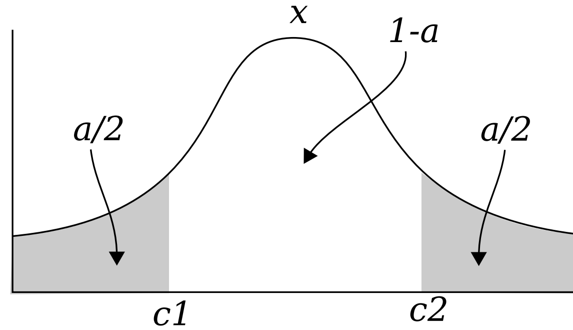

FIGURE 3.5: Accounting for errors in measurement of solver execution time x

{ 0.124, 0.142, 0.136, 0.131, 0.133 }

given a sample mean ($\bar{x}$) and sample standard deviation (s) of 0.133 and 0.006 respectively. Consulting a table for the t distribution, we find that when $a = 0.1$ and we require seven degrees of freedom, $t_{0.95;7} = 1.895$. Substituting these values into equation 3.4, we find

$$n = \left(\frac{z_{1-a/2}s}{e\bar{x}}\right)^2 = \left(\frac{1.895(0.006)}{0.035(0.133)}\right)^2 = 5.966 \quad (3.5)$$

Therefore, we need to make six measurements to be assured that there is a 90% chance that the true value is within this $\pm 3.5\%$ interval.

For completeness, the extra measurement yielded a value of 0.135; making the CPU time Alt-Ergo spent solving *first_last* the mean of the six measurements: 0.134 seconds.

3.3.2 Prover output

When a solver is sent a goal by Why3, it returns an answer A where A is one of $\{Valid, Invalid, Unknown, Timeout, Failure\}$. The following definitions of these answers are taken from the Why3 User Manual [28].

Valid	The goal is proved in the given context.
Invalid	The prover knows the goal cannot be proved.
Unknown	The prover has stopped its search.
Timeout	The prover has reached the time limit.
Failure	An error has occurred.

Fig. 3.6 shows the relative amount of $Valid/Unknown/Timeout/Failure$ answers from the eight SMT solvers (when given a timeout of 60 seconds) on the entire dataset of 1048 POs. For example, Alt-Ergo version 0.95.1 (leftmost bar) returned an answer of *Valid* for 590 POs, *Unknown* for 144 POs, *Timeout* for 300, and *Failure* for 14. Note that no tool returned an answer of Invalid for any of the 1048 proof obligations. We assume this is because the example programs are verified algorithms/data-structures and

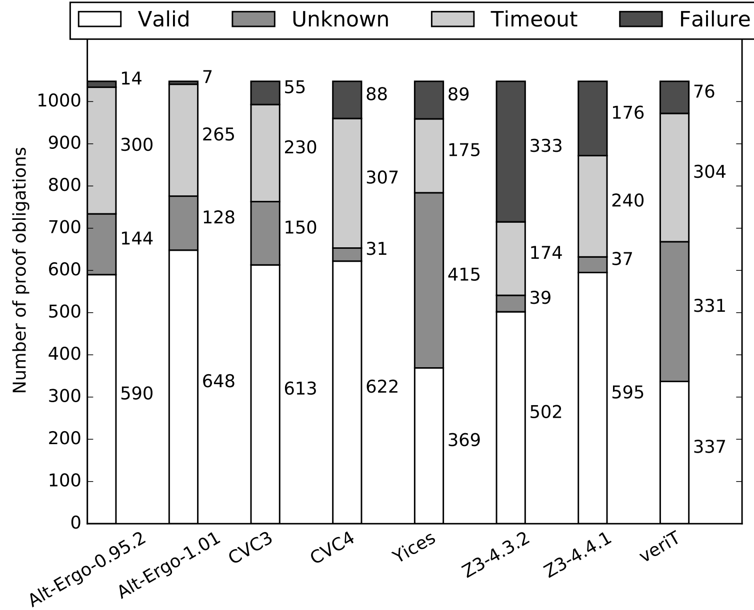

FIGURE 3.6: The relative amount of Valid / Unknown / Timeout / Failure answers from the eight SMT solvers (with a timeout of 60 seconds).

an answer of *Invalid* would indicate a violated pre-/post- condition or invariant. The limitations of our chosen dataset and other threats to validity are discussed later in the thesis (Sec. 6.4).

Often, goals that cannot be proved *Valid* or *Invalid* require inductive reasoning through the use of an interactive theorem prover such as Isabelle [93] or Coq [17]. Sometimes a splitting transformation needs to be applied in order to simplify the goals before they are sent to the solver. Where4 does not perform any transformations to goals other than those defined by the solver's Why3 driver file. In other cases, more time or memory resources need to be allocated in order to return a conclusive result. We address the issue of resource allocation in the next subsection.

Setting a timeout limit for measurement

A solver is said to return a *useful* result if it returns an answer of *Valid*, *Invalid* or *Unknown* when given a reasonable timeout limit. We justify this statement by making the observation that an answer of *Failure* usually means there is an error in logical translation of the PO. The failing solver is not a good choice for the particular logics required to return a *Valid* or *Invalid* answer. *Unknown* answers are usually identified as such very quickly therefore they do not impose a long waiting period for an answer and Where4 can call another solver with a minimal delay. By definition, *Timeout* results incur the maximal time penalty.

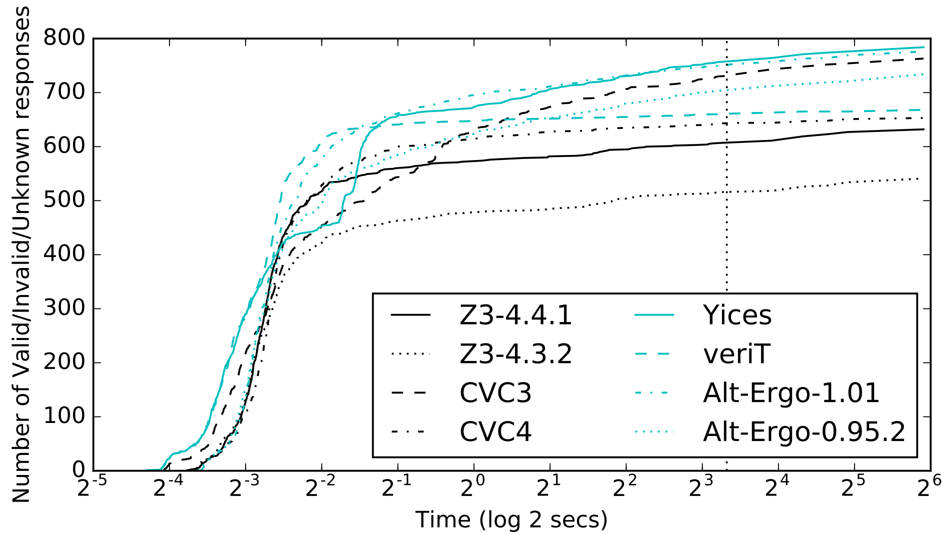

FIGURE 3.7: The cumulative number of *Valid/Invalid/Unknown* responses for each solver for the entire dataset of 1048 POs. The plot uses a logarithmic scale on the time axis for increased clarity at the low end of the scale.

Fig. 3.7 shows the number of *Valid/Invalid/Unknown* results for each solver when given a timeout value of 60 seconds. For example, the solid black line shows the useful results for version 4.4.1 of Z3. At 2^{-3} (0.125) seconds, Z3 returned 130 *Valid/Invalid/Unknown* results. This number increases sharply to 518 after 2^{-2} (0.25) seconds, before levelling off. If Z3 is given a time limit of 60 seconds, only 125 more useful responses are returned; giving a total of 632.

The value of 60 seconds was chosen as an upper limit, since this timeout value is not realistic for most software verification scenarios. Why3, for example, has a default timeout value of five seconds. By inspecting the plot of solver answers in Fig. 3.7, we can see that the number of useful answers returned levels off very quickly. From this observation we deduce that the likelihood of a *Timeout* answer potentially turning into a *Valid/Invalid* answer (if given more time) is minimal.

To establish a realistic timeout limit, we find each solver's "Peter Principle Point" [105]. In resource allocation for theorem proving terms, this point can be defined as the time limit at which more resources will not lead to a significant increase in the number of goals the solver can prove. By satisfying ourselves with being able to record 99% of the useful responses which would be returned after 60 seconds, a more reasonable threshold is obtained for each solver. To calculate this time value for each solver, we find the time at which 99% of the solver's total number of *Valid* / *Invalid* / *Unknown* responses have be returned. This threshold ranges from 7.35 secs (veriT) to 9.69 secs (Z3-4.3.2). Thus we chose a value of ten seconds (Fig. 3.7's *dotted vertical line*) as a representative, realistic timeout that gives each

solver a fair opportunity to return decent results.

3.4 Summary

In this chapter, we have introduced the SMT solvers, training and testing dataset, predictor and response variables used by the Where4 model. The choices made in regard to the SMT solvers were dictated to a large extent by Why3 and its selection of drivers. The dataset contains solutions to canonical SV challenges and is designed to demonstrate the specification and verification of data structures and algorithms fundamental to a wide range of applications, using the full capabilities of the Why3 system. The choice of predictor variables was influenced by the structural metrics introduced by McCabe (see Sec. 2.2), which have become established in the SE domain, and syntactic features similar to those used in TPTP library metadata. The response variables of *time* and *result* are similar to the scoring mechanism used for SV tools in the SV-COMP [19]. The response variables will be combined in the next chapter as a single value suitable for prediction by a variety of regression models.

After conducting the data preparation discussed in this chapter, we have collected particular information about each proof obligation:

- the time each of the eight solvers takes to return an answer when given a time limit value of ten seconds
- each solver's response (*Valid/Timeout/etc.*) for the same time limit
- a feature vector consisting of the twenty-eight structural and syntactic metrics listed in Fig. 3.2

The material presented in this chapter is critical to how the Where4 model functions and forms the basis for the experiments presented in the next chapter.

Chapter 4

Choosing a Prediction Model

This chapter will evaluate the effectiveness of a number of machine learning algorithms at predicting the most appropriate SMT solver to use on any (unseen) Why3 PO. Our goal in this thesis is to construct a "meta-solver" or *portfolio* solver which chooses from a range of tools in order to prove more goals than a single solver is capable of. We motivate the need for a portfolio solver with an analysis of our dataset in Sec. 4.1. More details about the type of prediction task chosen for the evaluation of ML algorithms is given in Sec. 4.2. A more detailed introduction to the six prediction algorithms (some of which were introduced previously in Sec. 2.3 and Table 2.2) is given in Sec. 4.3 before the results of their comparison is discussed. We make this comparison with reference to a number of theoretical strategies introduced in Sec. 4.4. We end this chapter with a more detailed look at the model chosen for use in the actual implementation of Where4.

Throughout this chapter, we use the terms "algorithms" and "models" to refer to different, but related, concepts. An "algorithm" refers to the general approach to prediction, as formalised to be applicable to a number of prediction scenarios. "Models" are trained instances of these algorithms which use specific parameters and data to make predictions for a single use case.

4.1 The benefit of portfolio-solving in Why3

Now that the SMT solvers to be supported by Where4 have been identified and an appropriate dataset for training and testing purposes has been chosen, we can make a preliminary and exploratory analysis of the behaviour of the SMT tools on the particular data. We aim to make a case for portfolio-solving as an effective method for discharging POs in the Why3 system.

We refer the reader to Table 4.1 which shows the results of running eight solvers on the example Why3 programs with a timeout value of ten seconds. The entire dataset is used in this case. WhyML files are modularised as one or more complete *theories* which can be used locally by other theories in the same file. The Why3 IVL identifies the *goals* which need to be proven in order for the theory (and in turn the entire file) to be verified as

TABLE 4.1: Results of running eight solvers on the example Why3 programs. Also included is a theoretical solver Choose Single, which always returns the best answer in the fastest time.

	File			Theory			Goal		
	# proved	% proved	Avg time	# proved	% proved	Avg time	# proved	% proved	Avg time
Choose Single	**48**	**37.5%**	**1.90**	**190**	**63.8%**	**1.03**	**837**	**79.9%**	**0.42**
Alt-Ergo-0.95.2	25	19.5%	1.45	118	39.6%	0.77	568	54.2%	0.54
Alt-Ergo-1.01	34	26.6%	1.70	142	47.7%	0.79	632	60.3%	0.48
CVC3	19	14.8%	1.06	128	43.0%	0.65	597	57.0%	0.49
CVC4	19	14.8%	1.09	117	39.3%	0.51	612	58.4%	0.37
veriT	5	4.0%	0.12	79	26.5%	0.20	333	31.8%	0.26
Yices	14	10.9%	0.53	102	34.2%	0.22	368	35.1%	0.22
Z3-4.3.2	25	19.5%	0.56	128	43.0%	0.36	488	46.6%	0.38
Z3-4.4.1	26	20.3%	0.58	130	43.6%	0.40	581	55.4%	0.35

TABLE 4.2: Breakdown of results in terms of triviality and hardness

	File	Theory	Goal
Trivial (all solvers can prove)	3	55	206
Hard (no solver can prove)	85	118	211
UNIQUELY-PROVABLE BY A SINGLE SOLVER			
Alt-Ergo-0.95.2	0	0	1
Alt-Ergo-1.01	6	12	25
CVC3	2	3	17
CVC4	1	6	33
veriT	0	0	0
Yices	0	0	2
Z3-4.3.2	0	0	0
Z3-4.4.1	0	0	2
Others: provable by at least two, but not all, solvers	32	104	551
TOTAL	129	298	1048

correct. WhyML theories containing assertions and lemmas can produce many goals while others which merely define types and helper functions can produce none. Our dataset of 128 WhyML files consists of 289 theories, which in turn generate 1048 goals. Aside from the number of each modular construct proved by the solver (*left sub-column*), the percentage this number represents of the total is given (*centre sub-column*) and the average time taken (in seconds) to prove each construct (as measured using the process described in Sec. 3.3.1) is given in the *right sub-column*.

We show the results of proving a WhyML file using its modular constructs. This method of verification is particularly useful when Why3 is run in batch mode from the command line. When using the Why3 IDE, it is more natural to prove programs by applying solvers to individual goals. This is because the environment separates WhyML programs into theories and goals automatically.

Table 4.1 also shows the results for a theoretical solver Choose Single. Choose Single is the best solver, from the eight SMT solvers measured, chosen on a per-goal basis. For example, if a file contains one theory which consists of three goals, and the best-performing solver is CVC4 for the first goal, Yices for the second, and CVC3 for the third, the result for Choose Single on that file is the sum of the results for CVC4 on the first goal, Yices on the second, etc. We define what is means for a solver to be the *best* in the next subsection.

The theoretical solver Choose Single shows the benefit of being able to use the most appropriate solver for each PO: 205 more goals are provable – an increase of 19.6% – over the best single solver (Alt-Ergo version 1.01) which can prove a total of 632. In total, 837 goals are provable by using a combination of the eight solvers – a figure that represents 79.9% of the 1048 goals. On average, Choose Single proves goals in a shorter amount of time than either version of Alt-Ergo or CVC3. The average time subcolumn provides an insight into how solvers which can prove relatively few goals, theories, or entire files – such as veriT or Yices – can be useful in a portfolio-solving context: such solvers often prove what they can in a very short amount of time (veriT takes an average time of just 0.12 seconds, for example, to prove each of five entire files) and can be the best choice of solver for those files, theories and goals.

The advantages of using multiple solvers are further illustrated by the results presented in Table 4.2. *Trivial* files, theories and goals (the first row of Table 4.2) are defined as those which can be proved by all eight solvers measured. Three files, 55 theories and 206 goals fall under the trivial category. Again, this table provides an interesting insight into the behaviour of all solvers: three of the five files proved by veriT are trivially provable by all solvers (which may explain the short amount of time taken, on average, to

prove the five files). The second row of this table shows the number of *hard* files, theories and goals. These are the tasks which no solver is able to prove entirely. As we have previously discussed in Sec. 3.3.2, these goals usually require the use of an ITP or splitting transformation to be discharged. Two hundred and eleven goals fall under this category.

The breakdown of the number of files, theories and goals which *only one* solver is able to prove is given in Table 4.2. The good performance of Alt-Ergo version 1.01 is obvious: six complete files can only be proved in their entirety by the solver. A higher number of goals require the use of CVC4 in order to be proved, however. Thirty-three goals (compared to twenty-five for the newer version of Alt-Ergo) can *only* be proved by this solver. The relatively low numbers for files, theories and goals provable by a single solver suggests that most goals can be proved by more than one tool, if they can be proved at all. Indeed, the second-to-bottom row of Table 4.2 shows that the majority of goals fall in this category.

4.1.1 The relative utility of solver responses

To make assertions about the relative performance of different solvers on the same goal, a definition of the relative *utility* of solver responses is required. Should a solver which returns an answer of *Valid* in five seconds be seen as "worse" than one that returns *Unknown* in half a second? Likewise, should the solver returning *Failure* after one second be penalised more severely than one returning *Timeout* after the maximum time limit?

We define an ordering for response utility as

$$\{Valid, Invalid\} > Unknown > \{Timeout, Failure\}$$

the reasoning being that a *Failure* response usually signals a fatal error in the logic encoding for that solver/goal pair, and the learning algorithm should be discouraged from choosing a failing solver for the particular goal characteristics in question. As discussed in Sec. 3.3.2 of the previous chapter, *Unknown* answers are returned quickly in general, and should not be penalised as much as *Timeout* responses. Solvers which reach the timeout limit are unlikely to return a *Valid* or *Invalid* response if given more time (illustrated clearly by Fig. 3.7). Solvers returning the same answer are ranked according to runtime – with faster solvers being preferred.

This method for defining relative performance has similarities to the scoring structure for ATPs competing in SV-COMP [18, 19], with some important differences. In SV-COMP, *Unknown* responses are given a neutral score of 0, the correct reporting of a property which does not hold ("true negative") is rewarded with smaller score (+1) than the correct reporting of

a property which is found to hold ("true positive" which scores $+2$). Likewise, the incorrect reporting of a property which does not hold ("false negative" or "false alarm") is punished less severely (with a score of -4) than the incorrect reporting of a property which does hold ("false positives" score -8). Although the notion of "false positive" and "false negative" responses is not applicable in the SMT domain (where tools are assumed to be sound), we follow the SV-COMP scoring scheme by awarding a "true positive" (or *Valid* response) a higher score than a "true negative" (or *Invalid* response). *Unknown, Timeout* and *Failure* responses are not treated separately by the SV-COMP scoring scheme – they all fall under the *Unknown* response category. We penalise poorly-performing solvers through the use of a *cost* function. The definition of the cost function we applied to solver results is given in Sec. 4.2.5 of this chapter.

4.2 Classification and regression

Machine learning prediction tasks can be separated into two categories: those involving the *classification* of a variable into discrete categories or classes, and those predicting a continuous-valued variable directly – *regression* tasks. This section will discuss some of the options considered when designing Where4's prediction task.

4.2.1 Predicting the single best solver

This option involves a multi-class classification task: the classes involved are the eight SMT solvers. Each PO is classified as belonging in one class. We reject this approach because some benefits associated with portfolio-solving are lost, since if the PO is misclassified, the performance of the portfolio solver suffers severely. For example, the single solver can return an answer of *Failure* with no other solver suggestion being made. In the best case, the predictions would be equivalent to the Choose Single theoretical solver.

4.2.2 Predicting the best *ranking* of solvers

Again, this option is a multi-class classification task. Instead of predicting a single solver, however, the task involves predicting the entire ranking of eight solvers. The benefit of obtaining a ranking is the flexibility it affords in calling SMT solvers sequentially or in parallel. If the first solver fails or returns an answer other than *Valid* or *Invalid*, the next best predicted solver is called, and so on.

With eight SMT solvers there are eight factorial (or 40,320) rankings which is far too many to be reasonable for a classification task. We rejected

this approach because in our dataset of 1048 POs many of these rankings could not possibly be represented. In fact, only 766 of the 40,320 rankings occur in our dataset. Such an unbalanced representation of classes cannot lead to accurate classification.

4.2.3 Predicting solver runtime and response separately

This approach involves two separate tasks, each predicting a characteristic of the solver's performance. One model would attempt to predict the response class (i.e. *Valid*, *Invalid*, etc.) while another would attempt to predict the solver runtime. The former task is a multi-class classification task with five classes, while the latter is a multi-output regression task. This method has the advantage of affording the user flexibility in how to choose the ultimate ranking: for example, if fast responses are preferred over *Valid* answers. This flexibility comes at the price of complexity, however: two accurate predictors, a classifier and regressor, are required instead of one.

4.2.4 Combining the prediction of solver response and runtime

This option uses a cost function to combine the two solver response variables as a single real-valued number which is used for ranking the solvers. This is a multi-output regression task: a cost prediction is made for each solver individually. The individual predicted values are sorted in increasing order to produce a ranking of decreasing "solver utility".

This approach shares most of the advantages of predicting solver response and runtime separately: each solver's actual behaviour is predicted directly rather than relying on the relative behaviour of all eight *together* constituting a single class. It is relatively simple to change the cost function, although it can not be done "on-the-fly" in the same manner as the previous approach allows. A change to the cost function requires re-training the model. The major benefit is having one single model to predict both the solver response and the runtime. This is the prediction approach chosen for Where4 in this thesis.

4.2.5 The Cost Function

The use of a cost function is inspired by the first version of the SATZilla portfolio solver [114]. The cited paper describes the prediction of a *performance score*. This differs from most portfolio approaches which aim to predict a solver's runtime directly.

The cost function should reflect the ordering of solver utility defined in Sec. 4.1.1: penalising poorly-performing solvers while ensuring *Valid* and *Invalid* responses incur the lowest cost. The following simple function

allocates a cost to each solver S's response $\langle answer, time \rangle$ to each goal G:

$$cost(S,G) = \begin{cases} time_{S,G}, \text{ if answer}_{S,G} \in \{Valid, Invalid\} \\ time_{S,G} + \text{timeout}, \text{ if answer}_{S,G} = Unknown \\ time_{S,G} + (\text{timeout} \times 2), \text{ if answer}_{S,G} \in \{Timeout, Failure\} \end{cases} \tag{4.1}$$

Thus, to penalise the solvers that return an *Unknown* result, the timeout limit is added to the time taken, while solvers returning *Timeout* or *Failure* are further penalised by adding double the timeout limit to the time taken. A response of *Failure* refers to an error with the back-end solver and usually means a required logical theory is not supported. This function ensures the best-performing solvers always have the lowest costs. A ranking of solvers for each goal in order of decreasing relevance is obtained by sorting the solvers by cost in ascending order.

Other cost functions were trialled before this formulation was decided upon. Previous methods allocated a cost penalty (based on the answer) and calculated the solver cost as the Euclidean distance from the origin to point defined as this penalty and the solver's time. Eg: $cost(S,G) = dist((time_{S,G}, penalty_{ans}), (0,0))$. Where the penalties associated with each response are:

$$penalty_{ans} = \begin{cases} 0 \text{ if ans} = Valid \\ 5 \text{ if ans} = Invalid \\ 10 \text{ if ans} = Unknown \\ 15 \text{ if ans} = Timeout \\ 20 \text{ if ans} = Failure \end{cases} \tag{4.2}$$

The problem with this formulation is that the penalty values are arbitrarily determined and bear no relation with the timeout value. It is therefore not guaranteed that our defined *relative utility* of solvers is respected by using the penalties of Eq. 4.2. This issue led us to re-formulate the cost function to incorporate the timeout value as follows:

$$cost(S,G) = \begin{cases} time_{S,G}, \text{ if answer}_{S,G} \in \{Valid, Invalid\} \\ time_{S,G} + \text{timeout}, \text{ if answer}_{S,G} = Unknown \\ dist((time_{S,G}, \text{timeout}), (0,0)), \text{if answer}_{S,G} \in \{Timeout, Failure\} \end{cases} \tag{4.3}$$

which only makes use of the Euclidean distance for *Timeout* and *Failure* responses. As pointed out by a reviewer of an early version of this work [63], in certain circumstances a *Timeout* response may have a lower cost than an *Unknown* response. This is the case when the *Unknown* response is returned in a time >= 0.42 times that of the time taken to return a response of *Timeout*. This property of the triangle inequality was account for in the

TABLE 4.3: Result of eight solver executions on *first_last*

SOLVER	VERSION	RUNTIME (SECS)	RESPONSE	COST	RANK POSITION
Alt-Ergo	0.95.1	0.134	Unknown	10.134	2
Alt-Ergo	1.10	0.170	Unknown	10.171	3
CVC3	2.4.1	0.356	Valid	0.356	1
CVC4	1.4	0.173	Unknown	10.173	4
veriT	201506	10.109	Timeout	30.109	5
Yices	1.0.38	10.161	Timeout	30.161	8
Z3	4.3.2	10.115	Timeout	30.115	6
Z3	4.4.1	10.131	Timeout	30.131	7

final formulation of the cost function (Eq. 4.1).

Example: using the cost function to obtain a ranking for *first_last*:
To illustrate the effect of applying our cost function, we return to the *first_last* goal introduced in the previous chapter. Table 4.3 lists the results of executing the eight solvers on the goal, as measured using the method described in Sec. 3.3, with a time limit of ten seconds. The derived cost value for each solver is used as its dependent variable in the prediction models compared in Sec. 4.3 and Where4's actual implementation.

4.3 Choosing the most effective algorithm for rank prediction

Fig. 4.1 illustrates a high-level view of the process used to compare and evaluate the various ranking strategies in this section. The solver timings and results are combined as a single variable using the cost function defined in Sec. 4.2.5. The value returned by this function is the response variable for a number of ML algorithms; with the statically-extracted features acting as the predictor variables.

A four-fold cross validation of this data is used to evaluate the models. In this method, the data is split into quarters. Four models are effectively trained on different datasets by holding a different quarter back for model evaluation each time. The final evaluation is based on the model's average performance over the four instances. The general *K-fold* cross validation method allows the use of the entire dataset for model evaluation and training (rather than holding a portion of the data back solely for model evaluation use – a technique known as "hold-out" validation). The number of folds used (four) was chosen to reflect the percentage of the total dataset (75%) used for training. The folds were stratified (i.e. ensuring a representative frequency of values was contained in each split) according to the

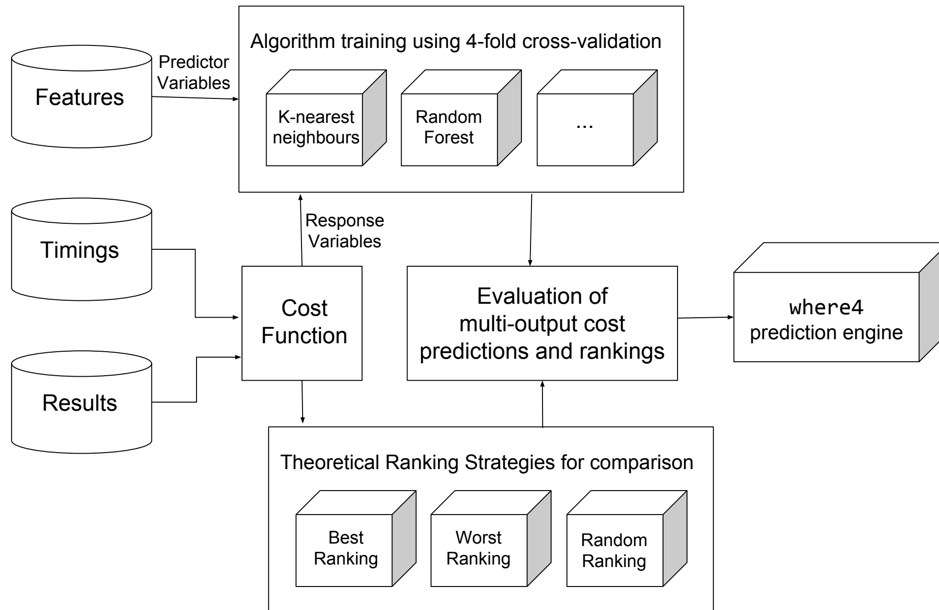

FIGURE 4.1: Overview of the process used to derive the Where4 prediction model

best performing solver for each instance. This technique was used to overcome the difficulties of stratification for regression tasks where values may appear infrequently.

4.3.1 The ML algorithms we used for model training

The ML algorithms compared in this section have been briefly introduced in Sec. 2.3. We do not compare Naïve Bayes which does not support regression tasks. In this section, we go into more detail about how the algorithms operate and the specific variants and parameters chosen for use during this comparison. We used the Sci-Kit Learn [95] Python implementation for all algorithms. The consistent API is well-designed, the library is well documented and it integrates well with other Python tools for scientific computing (such as NumPy and the Pandas data structure library).

ML Algorithm 1: Support Vector Machines

Support Vector Machines (SVMs) [46] are based on a concept of similarity between training instances. During training, a number of separating hyperplanes are determined. The idea behind SVMs is to ensure the distance from this hyperplane to any training instances closest to it is maximal. This process is known as “maximising the margin” and is designed give the model the best chance at classifying unseen instances correctly. The set of training instances closest to the margin is called the *support vector* as they can be thought of as “holding up” the hyperplane.

A number of kernel functions can be used as a similarity measure during training. We chose to use the *Radial Bias Function* (RBF) kernel as we can not assume a linear relationship between our features and cost value. The RBF kernel requires the use of two hyperparameters to control how flexible the model is (i.e. how closely it fits the training data). We used a grid search technique [66] to identify the C (hyperplace smoothness) and $gamma$ (the radius of influence for each individual training instance) parameters.

In the multi-output case, each cost value has to be predicted individually for each solver – requiring the use of eight SVMs each tuned to different parameters on a per-solver basis. The benefit of using SVMs is their ability to give good results in large-dimensional spaces; even with large datasets.

We followed Hsu and Lin's [67] recommendations regarding the use of SVMs (which excel at *binary* classification) for multi-output regression tasks.

ML Algorithm 2: Decision Trees

Decision trees recursively construct a binary tree by splitting the data on the feature and threshold which create the most distinct partitions [97]. This notion of "distinctiveness" is defined using either a measure of entropy or information gain. Features of new instances are queried according to the thresholds specified by the non-leaf nodes, with leaves consisting of (one or many) predictions. The original implementation required that features be categorical (Quinlan's ID3 algorithm). Sci-kit Learn uses the CART [36] algorithm which supports numerical features and outputs for regression tasks.

Decision trees are a transparent and powerful method for classification and regression. Since the 1960s, decision trees have been successfully deployed in many domains. In the intervening years, techniques have been developed to improve the accuracy and generalisability of decision trees: such techniques include the pruning of *if-then* branches created by sequences of splitting nodes. Termination conditions can be used to limit the depth of the tree and make each leaf node account for a minimum number of examples in the training data. We chose this last technique as a method to prevent our decision tree from overfitting training data: a minimum of five training instances had to be described by the *if-then* rule associated with the leaf. By using this constraint, non-leaf nodes are converted into leaves if, when split, they would have produced leaves of size less than five. This relatively small number (five) was chosen as a compromise: leaves of this size allow the decision tree to generalise better than trees with leaves accounting for fewer training instances. At the same time, the leaves remain small enough to allow the tree to utilise many of the instance's features for its characterisation (i.e. the tree's depth does not become too small).

ML Algorithm 3: Random Forests

Random forests [35] were created as a response to decision tree's tendency to overfit training data – leading to poor generalisation results. As the name suggests, this technique involves creating many decision trees, with each tree trained on a random subset of the training data and/or restricted to using a subset of the features for use in splitting nodes. Random forests are an example of an *ensemble* method which uses multiple weak predictors to strengthen the overall prediction.

For classification problems, each tree "votes" on an instance's class. In the regression case, all trees' predictions are averaged to determine the forest's ultimate prediction.

Similar to our use of decision trees, we limited the depth of each tree by specifying a minimum size of five for each leaf node. For this initial algorithm comparison we used 100 trees in the random forest. This is the default Sci-Kit Learn implementation.

ML Algorithms 4 & 5: Linear and Ridge Regression

Linear regression attempts to predict the parameters of a function which fits the input features to the output variable. It expects that the output variable be a linear combination of the input variables. We used the Ordinary Least Squares formulation which attempts to minimise the sum of squared error for the set of training instances to the output variable.

Ridge regression [65] is a related algorithm which attempts to be more robust to violations in the input variables' independence.

ML Algorithm 6: k-Nearest Neighbours Clustering

K-Nearest Neighbours (k-NN) clustering, like SVMs, belong to the analogical family of ML algorithms which rely on a similarity measure to group training instances (we used the standard Euclidean distance). The idea is to create k clusters of training instances which are "most similar" to each other in terms of features – with each feature acting as a dimension. In both the k-NN and SVM algorithms, instances must undergo a normalisation process to scale features. This preprocessing step is designed to avoid dominance of one feature over another (due to differences in scale) in distance-based algorithms. In contrast to SVMs, most clustering algorithms (k-NN included) do not scale well in high-dimensional spaces.

Typically used for classification tasks, the k-NN algorithm can be adapted for regression by calculating the target/response variable as the average of the k nearest instances in the training data. By default, Sci-Kit

Learn sets $k = 5$: each instance is compared to the five most similar instances (in terms of input variables) in the training set. We used a modification: training instances "closer" to the unseen instances are weighted more than those further away in the cluster when computing the target variable. This modification makes the algorithm more robust to noisy training data.

4.3.2 Experimental Configuration

In the previous chapter, we described our dataset as consisting of 1048 Why3 POs. We now split this dataset into two disjoint subsets: the *training* and *testing* sets. The rest of this chapter uses only the *training* set while the *testing* set is used in our final evaluation of Where4 in Chapter 6. The training set represents 75% of the entire dataset: 96 files, 212 theories and 785 goals. The data was split on a per-file basis to ensure that the no PO in the training set belonged to the same theory or file as a PO in the test set.

Beside the standard implementation of the prediction algorithm, two variations were evaluated: cost discretisation and instance weighting. By discretisation we describe the process to transform a continuous-valued variable into one of a finite number of values. It usually involves dividing the continuous variable by some small empirically-tested number. Discretisation allows algorithms which perform better when given a smaller number of discrete options for prediction to be identified. We chose 2.5 to be a discretisation divisor for the response variable (i.e. each solver's cost). In choosing this value, two factors must be balanced: the discretisation error inherent in the process should be minimised while allowing only a relatively small number of possibilities for prediction.

The other technique we applied during model evaluation was the weighting of training instances[1]. Weighting is standard practice in supervised machine learning: each instances's weight was defined as the standard deviation of solver costs for the PO in question. This function was designed to give more importance to instances where there was a large difference in performance among the solvers; thereby de-emphasising trivial POs provable by most or all solvers, and empirically hard instances for which all solvers fail or time out.

4.4 Ranking strategies

The ranking strategies introduced in this section are not directly comparable to single solvers or to theoretical solvers such as Choose Single. The purpose of defining these strategies is to provide a basis by which we can compare and evaluate the solver rankings predicted by the ML models.

[1] Apart from K-Nearest Neighbours: the Sci-kit Learn implementation does not support instance weighting.

The Best Ranking and Worst Ranking strategies use the empirical measurements to construct rankings on a per-goal basis, while the Random Ranking strategy uses the same set of rankings for each goal. We refer to the following strategies as *theoretical* as they assume knowledge of the actual behaviour of the solvers for the goal in question – knowledge which is impossible to have prior to execution in a real-world verification scenario.

Algorithm 1 describes the process to return answers and runtimes from solver rankings. This process is used in our model evaluation (Sec. 4.5). Essentially, the runtime of next best solver (according to a given ranking) is added to the cumulative total until an answer of *Valid* or *Invalid* is returned or the array of solvers has been exhausted. When solver answers are compared in the `if` statement, the ordering of response utility introduced in Sec. 4.1.1 (with *Timeout* responses preferred to *Failure*): $\{Valid, Invalid\} > Unknown > Timeout > Failure$. If both *Valid* and *Invalid* responses were recorded for the same PO, it would indeed signal a soundness bug existed in one of the solvers. This issue did not arise.

```
Input: Solvers {S_1, ..., S_n} sorted by cost (predicted, actual, or
       random)
Output: ⟨A, T⟩ where A = the best answer from the solvers; T = the
        cumulative time taken to return A
begin
    /* initialisation                                          */
    A ← Failure
    T ← 0
    i ← 1
    while A ∉ {Valid, Invalid} ∧ i ≤ n do
        /* A_S = the answer returned by solver S_i             */
        A_S ← Answer(S_i)
        /* add solver S_i's time to the cumulative
           runtime                                             */
        T ← T + Time(S_i)
        if A_S > A then
            /* S_i's answer is better than the current
               best answer                                     */
            A ← A_S
        end
        i ← i + 1
    end
    return ⟨A, T⟩
end
```

Algorithm 1: Returning answers and runtimes from solver rankings

4.4.1 Best Ranking

The Best Ranking is the one derived from sorting the solvers in terms of increasing cost. Referring to the solver costs listed in Table 4.3, the Best

Ranking for this goal is CVC3 > Alt-Ergo-0.95.1 > Alt-Ergo-1.01 > CVC4 > veriT > Z3-4.3.2 > Z3-4.4.1 > Yices. As the first-choice solver according to this ranking (CVC3) returns an answer of *Valid*, no other solver will be called after the first returns its answer. Thus for the *first_last* goal, the Best Ranking strategy returns an answer of Valid in 0.356 seconds.

It is important not to confuse the Best Ranking strategy with the theoretical solver Choose Single introduced in Sec. 4.1. Choose Single refers to a *single* solver (and hence is directly comparable to the other eight SMT solvers in Table 4.1), while Best Ranking refers to a *ranking* of all eight solvers. Choose Single is equivalent to using the top-ranking solver from Best Ranking and stopping.

4.4.2 Worst Ranking

Ranks returned by the Worst Ranking strategy are the inverse to those from the Best Ranking. For the *first_last* goal, Yices will be the first solver called by this strategy. As Yices returns an answer other than *Valid* or *Invalid*, the next solver in the ranking (Z3 version 4.4.1) will be called, and so on. The user of this strategy would have to wait until the eighth solver, but a *Valid* answer would eventually be returned. For the example goal, the Worst Ranking strategy returns an answer of *Valid* in 41.349 seconds.

Both the Best Ranking and Worst Ranking strategies provide important upper and lower bounds for any trained model's runtime. If a goal is provable by any of the eight solvers, all of our theoretical strategies will be able to prove it. The difference between these strategies is how long the goal takes to be proven. Worst Ranking therefore acts as a method to obtain the *worst case* runtime (which would in fact be equal to that of Best Ranking for POs which cannot be proved by any of the eight solvers). This point is important to bear in mind when evaluating prediction models (and their implementation in Where4), as a seemingly ineffective ordering may be equivalent to the best ordering of solvers.

4.4.3 Random Ranking

The Random Ranking strategy differs from the previous two because it does not refer to *one* ranking per goal but to *all possible* rankings for every goal. The set of all possible rankings is the same for all goals, but the runtime and result obviously vary. As previously mentioned, there are eight factorial (or 40,320) possible solver rankings. The runtime for each of the 40,320 rankings is calculated and the mean of these times is returned by Random Ranking. As mentioned in relation to Worst Ranking, for any goal the result returned by all rankings (of the same eight solvers) is the same.

For the *first_last* goal, the Random Ranking strategy returns an answer of *Valid* in an average time of 20.853 seconds.

4.4.4 Quantifying Solver Contributions Using Ranking Strategies

In addition to the findings presented in Sec. 4.1.1 which measured the goals uniquely provable by each solver, this section quantifies how valuable each individual solver is to a ranking of solvers. We follow a method described by Xu et al [113]: each constituent solver A's value is measured by observing the performance of the theoretical best solver which does not include A. Hence, A's *marginal contribution* to Best Ranking can be defined as the difference between Best Ranking's cumulative cost excluding A: $cost(\text{Best Ranking}_{-A})$; and its cumulative cost including A: $cost(\text{Best Ranking})$.

$$contrib(A) = cost(\text{Best Ranking}) - cost(\text{Best Ranking}_{-A})$$

We define the cost of a ranking to be the cumulative cost for each of its constituent solvers: for any given PO, we accumulate the cost of each solver called until an answer is returned according to Alg. 1. To ensure $cost(\text{Best Ranking}) \leq cost(\text{Best Ranking}_{-A})$, the two rankings need to have the same number of solvers. Thus, we exclude the lowest ranking solver from Best Ranking for these purposes. This accounts for POs unprovable by any solver (*Hard* in terms of the data presented in Table 4.2) which have a relatively high cost for all solvers. Including these costs when comparing rankings would bias the results towards the rankings with fewer solvers: the Best Ranking excluding A will always have a lower cost than the ranking including A for *Hard* POs.

For each solver in the set of solvers S, we normalise its contribution by dividing it by the sum of all solver contributions and multiplying by 100:

$$contrib_{norm}(A) = \frac{contrib(A)}{\sum_{i=1}^{|S|} contrib(S_i)} \times 100$$

Figure 4.2 shows the normalised contribution statistics for each solver in the portfolio. Each solver's percentage represents the impact its exclusion has on Best Ranking's accumulated cost across the entire dataset, relative to the other solvers. It is unsurprising that Alt-Ergo-1.01 and CVC4 contribute most to the portfolio: these solvers can prove the most uniquely-provable POs (according to Table 4.2). It is somewhat unexpected that veriT and Yices contribute more than either version of Z3, given that they can prove less goals overall (Table 4.1). This implies that (i) Z3 does not perform significantly better than other solvers for provable goals, (ii) veriT and Yices

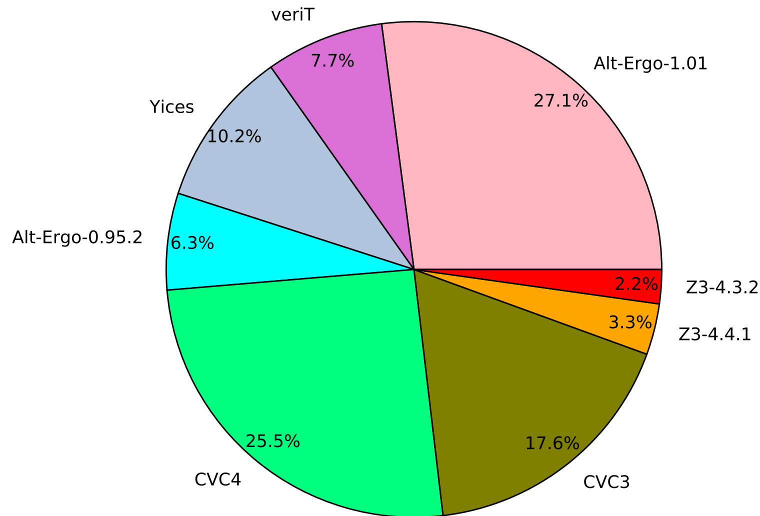

FIGURE 4.2: Relative solver portfolio contributions

prove goals in a very short time and (iii) for goals not provable by any of these solvers, veriT and Yices return a better answer in a shorter time.

It is logical that earlier versions of solvers contribute less than more recent versions. Alt-Ergo-0.95.2 contributes significantly less than Alt-Ergo-1.01: we deduce that the earlier version takes more time to return a worse answer for *Hard* POs than the update. Nevertheless, including multiple versions of solvers improves prediction accuracy and ultimately makes Where4 a more flexible tool for the user.

4.5 Predictor Selection Results

In Table 4.4 we list the cross-validation results for all six trained prediction models (with discretised and/or weighted variants where applicable). We compare these with the three theoretical ranking strategies introduced in Sec. 4.4. For each of the five Evaluation Criteria EC1-5, the best-performing algorithm is marked in bold with an asterisk.

4.5.1 EC1: Time

The first result column, labelled **Time (secs)**, shows the average time taken for the ranking of solvers returned by the model/strategy to return a *Valid* / *Invalid* response (if such a response was in the set of solver answers – see Algorithm 1).

If there is no *Valid* / *Invalid* response in the set, the time will be the same for any ordering of the solvers (ie. all eight will be called). We do not ignore the time taken for the response to be returned.

4.5.2 EC2: R^2 Score

The second numeric column of Table 4.4 shows the R^2 score (or coefficient of determination), which is an established evaluation criterion. It measures how well regression models can predict the variance of dependent / response variables. The maximum R^2 score is 1 but the minimum can be negative. Note that the theoretical strategies return rankings rather than individual solver costs. For this reason, R^2 scores are not applicable.

The R^2 score is calculated by the following formula

$$R^2 = 1 - \frac{\Sigma(y_i - \hat{y})^2}{\Sigma(y_i - \bar{y})^2} \tag{4.4}$$

which can be interpreted as “the sum of squared error of the predictions divided by the sum of the squared mean". In Formula 4.4, $y_i - \hat{y}$ denotes the distance of the predicted value $\hat{y}$ from the actual value y for data point i. $\bar{y}$ is the mean value from the set of actual values (which in our case is a two-dimensional array of solver scores). We used the Sci-Kit Learn implementation of this formula, which allows $\bar{y}$ to be set to the weighted average of the outputs for multi-output instances. We applied uniform weighting to all outputs; meaning the variance of the individual solvers' scores was not considered.

4.5.3 EC3: Normalised Distributed Cumulative Gain

The third numeric column of Table 4.4 shows the Normalised Discounted Cumulative Gain (**nDCG**), which is commonly used to evaluate the accuracy of rankings in the search engine and e-commerce recommender system domains [73]. Here, emphasis is placed on correctly predicting items higher in the ranking. For a general ranking of length p, it is formulated as:

$$nDCG_p = \frac{DCG_p}{IDCG_p} \quad \text{where} \quad DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1)} \tag{4.5}$$

where rel_i refers to the relevance of element i with regard to a ground truth ranking, and we take each solver's relevance to be inversely proportional to its rank index. In our case, $p = 8$ (the number of SMT solvers). The DCG_p is normalised by dividing it by the maximum (or *idealised*) value for ranks of length p, denoted $IDCG_p$. As our solver rankings are permutations of the ground truth (making $nDCG$ values of zero impossible), the values in Table 4.4 are further normalised to the range [0..1] using the lower $nDCG$ bound for ranks of length eight – found empirically to be 0.4394.

We illustrate our use of this criterion with a small example. Take a ground truth ranking of eight items sorted in decreasing relevance as

$$A > B > C > D > E > F > G > H.$$

TABLE 4.4: Predictor Selection Results. * indicates the best result among the prediction models

	Discretised	Weighted	Time (secs)	R^2	nDCG	MAE	Reg-error
Best Ranking			12.59	-	1.00	0.00	0.00
Random Ranking			19.06	-	0.36	2.62	50.77
Worst Ranking			30.23	-	0.00	4.00	94.57
Decision Tree	✗	✗	16.37	0.09	0.49	2.08	45.00
	✓	✗	15.90	-0.18	0.43	2.27	42.65
	✗	✓	15.81	-0.06	0.48	2.09	41.14
	✓	✓	15.75	-0.17	0.43	2.25	41.80
k-Nearest Neighbours	✗	✗	15.40	0.17	***0.55**	***1.90**	40.65
	✓	✗	15.49	-0.19	0.44	2.26	41.47
Linear Regression	✗	✗	15.60	-0.11	0.41	2.45	49.70
	✓	✗	15.74	-0.30	0.41	2.46	49.12
	✗	✓	15.78	-0.32	0.41	2.43	49.72
	✓	✓	15.80	-0.26	0.41	2.45	50.12
Support Vector Regressor	✗	✗	15.59	0.15	0.49	2.31	46.17
	✓	✗	15.61	-0.23	0.43	2.31	43.17
	✗	✓	15.35	-0.00	0.46	2.26	41.65
	✓	✓	15.53	-0.19	0.44	2.26	43.17
Random Forest	✗	✗	14.99	***0.28**	0.48	2.09	39.63
	✓	✗	15.04	-0.18	0.47	2.13	39.20
	✗	✓	***14.88**	0.20	0.49	2.10	***38.26**
	✓	✓	15.03	-0.15	0.49	2.11	38.66
Ridge Regression	✗	✗	15.55	-0.09	0.41	2.46	49.85
	✓	✗	15.65	-0.30	0.41	2.47	49.41
	✗	✓	15.58	-0.08	0.40	2.49	50.12
	✓	✓	15.64	-0.22	0.40	2.50	50.69

TABLE 4.5: Illustration of various nDCG scores (normalised for permutations of length eight)

PERMUTATION	NDCG SCORE
ABCDEFGH	**1.0**
ABCDEF*HG*	0.9998
ABCD*FE***GH**	0.9989
AB*DC***EFGH**	0.9899
*BA***CDEFGH**	0.7848
ABCD*HGFE*	0.9950
*DCBA***EFGH**	0.3809
HGFEDCBA	0.0

We take each item's *relevance* to be inversely-proportional to its rank position:

$rel_A = 8, rel_B = 7, rel_C = 6, rel_D = 5, rel_E = 4, rel_F = 3, rel_G = 2, rel_H = 1.$

The predicted ranking of these items is

$$B > A > C > F > D > E > G > H.$$

Applying Equation 4.5 to these values gives us an $nDCG$ value of

$$\frac{127}{1.0} + \frac{255}{1.58} + \frac{63}{2.0} + \frac{7}{2.32} + \frac{31}{2.58} + \frac{15}{2.81} + \frac{3}{3.0} + \frac{1}{3.17} = 341.053.$$

It can be shown that $IDCG_8 = 389.591$ which makes the un-normalised $nDCG$ value for the predicted ranking 0.875. Using the lower bound of 0.4394 to normalise this value to the range $[0, 1]$ (accounting for permutations of length 8) gives us a final $nDCG$ score of 0.778.

The nDCG metric is best understood in relation to other rankings in the same problem domain: a score of say, 0.5, is meaningless in isolation. For this reason, we provide Table 4.5 so that the reader may gain an intuitive understanding of this metric as we use it.

4.5.4 EC4: Mean Average Error

Table 4.4's fourth numeric column shows the MAE (Mean Average Error) – a ranking criterion which can also be used to measure string similarity. It measures the average distance from each predicted rank position to the solver's index in the ground truth.

Using the same predicted and actual rankings as the $nDCG$ example, we calculate the MAE to be:

$$MAE = \frac{1 + 1 + 0 + 2 + 1 + 1 + 0 + 0}{8} = 0.75$$

The Mean Average Error is a good measure of overall ranking accuracy, but it fails to take into account that in many use cases – including Where4 – it is often the highest-ranking items that are the most important to predict accurately.

4.5.5 EC5: Regression Error

The rightmost column of Table 4.4 (labelled **Reg. error**) lists the average regression error per instance. We illustrate the calculation of the regression error with a small example. For two testing instances, the predicted and actual costs for three solvers are:

$$\begin{pmatrix} 0.5 & 10.0 & 5.0 \\ 0.4 & 5.5 & 10.0 \end{pmatrix} = \text{predicted}; \begin{pmatrix} 0.6 & 9.0 & 8.0 \\ 0.7 & 9.5 & 7.0 \end{pmatrix} = \text{actual}$$

which gives a per-instance regression error of

$$\begin{pmatrix} 0.1 + 1.0 + 3.0 \\ 0.3 + 4.0 + 3.0 \end{pmatrix} = \begin{pmatrix} 4.1 \\ 7.3 \end{pmatrix}$$

taking the average of the two instances' regression error, 4.1 and 7.3, gives 5.7. It is this average error between predicted cost and actual cost for eight solvers which is listed in the rightmost column.

The regression error should be treated with a note of caution, however, as the relative ranking of solvers – which we argue is the most important prediction – is not explicitly taken into account. This qualification also applies to the MAE and R^2 score criteria. Note that when the predictions and actual costs are sorted in increasing order the best-performing solver is predicted correctly for both instances in the previous example. The entire ranking of solvers is predicted correctly in the first instance.

4.5.6 Properties of multi-output problems

An interesting feature of all the best-performing models in Table 4.4 – Random Forests, K-Nearest Neighbours, Decision Trees – is their ability to predict *multi-output* variables. In contrast to Support Vector Machines, for example, which must predict the cost for each solver individually, an algorithm which supports multi-output problems can predict each solver's cost simultaneously. Not only is this method more efficient (by reducing the number of estimators required), but it has the ability to account for the correlation of the response variables. This is a useful property in the software verification domain where certain goals are not provable and others are trivial for SMT solvers. Multiple versions of the same solver can also be expected to have highly correlated scores.

TABLE 4.6: Relative importance of each feature in the Where4Random Forest model

FEATURE	IMPORTANCE (%)	FEATURE	IMPORTANCE (%)
`func`	19.63	`divisor`	2.66
`avg-op-arity`	11.05	`n-ops`	2.48
`int`	8.96	`not`	1.15
`forall`	7.19	`if`	1.07
`and`	6.02	`case`	0.99
`depth`	5.70	`wild`	0.84
`var`	4.84	`false`	0.37
`impl`	4.72	`or`	0.36
`size`	4.31	`iff`	0.33
`let`	4.03	`eps`	0.13
`n-quants`	3.70	`true`	0.12
`n-preds`	3.27	`exists`	0.07
`zero-ar`	3.03	`float`	0.00
`n-branches`	2.97	`as`	0.00

A more general survey of multi-output regression and how various ML algorithms can either be adapted or extended for this use case is provided by Borchani et al. [31].

4.5.7 The chosen model

After inspecting the results for all trained models (summarised in Table 4.4), we can see that Random Forests [35] perform well, relative to other methods. They score highest for three of the five criteria (shown in bold with an asterisk) and have generally good scores in the others. Based on these results, we selected Random Forests as the choice of algorithm to use in Where4. We chose to implement a Random Forest model which does not use discretisation or instance weighting. Inspecting the results from the entire range of algorithms shows that, in general, the best-performing models do not use these techniques.

The k-Nearest Neighbours model, while coming top for two important criteria, scored poorly in the **Time** category. This result leads us to conclude that k-Nearest Neighbours can often predict good rankings but the instances on which it performs badly are very expensive in terms of time. Overall, the Random Forest model is the most consistent and reliable predictor.

Before discussing the implementation of Where4in the next chapter, we shall investigate the properties of the Random Forest model after it was fit on the entire training dataset.

Table 4.6 shows the relative importance of each predictor variable for decision-making in the Random Forest model. Every time a split of a node

is made on a feature the impurity criterion for the two descendent nodes is less than the parent node. This criterion is called the *Gini impurity* [35]. The percentages correspond to the proportion of the *Gini* decrease accounted for splitting by each feature. This data is part of Sci-Kit Learn's implementation of Decision Trees and Random Forests.

It can be seen that the number of functions in each PO is by far the most important factor in determining each solver's cost, with the average branching factor of functions and other operators also being important. It is somewhat surprising that the number of integer constants is one of the more important factors while the number of floating point constants is relatively unimportant. This may be due to the fact that there are more integer-based POs in our dataset than those which use floating point numbers. This reasoning may also account for the fact the universal quantifiers (`forall`) rank higher than existential (`exists`) quantifiers.

4.6 Summary

This chapter began by motivating the use of portfolio-solving in the Why3 system by showing that more files, theories and goals can be proven by using the best-performing solver on a per-goal basis. We then gave a tour of the prediction approaches considered for use in Where4. We decided to use a regression method to predict each solver's *cost* – a value designed to account for solver response as well as the time taken to return the response.

A number of ranking strategies and evaluation criteria were described in Sec. 4.4 and Sec. 4.5 respectively. In this chapter, we used them to compare various trained models and found Random Forests to be the best choice for implementation in Where4.

Some properties of the trained Random Forest model were then discussed. The choice of prediction model has an obvious importance on Where4's implementation and success as a predictor. We give more details of this implementation in the next chapter.

Chapter 5

OCaml Implementation

This chapter will give some details of the choices we made when implementing the Where4 tool. We decided to make Where4 available as a stand-alone tool on the command-line as well as through the Why3 system by imitating an orthodox SMT solver.

The implementation of Where4 makes use of various techniques and heuristics encountered when researching related premise selection and portfolio solving tools such as those described in sections 2.3.1 and 2.3.2 of the Literature Review. For example, Where4's interaction with Why3 is inspired by Sledgehammer's MaSh [24] tool. MaSh uses machine learning to suggest premises based on a large corpus of learned theorems, allowing POs to be proved automatically by ATP and SMT tools. We aspired to Sledgehammer's "zero click, zero maintenance, zero overhead" philosophy in this regard: Where4 should not interfere with a Why3 user's normal work-flow nor should it penalise those who do not use it.

One heuristic we implemented was to call the highest ranking solver installed on the user's system from the following static ranking: Alt-Ergo-1.01 > CVC4 > CVC3 > Z3-4.4.1 > Alt-Ergo-0.95.1 > Z3-4.3.2 > Yices > veriT. We derived this ranking from the total number of POs each solver could prove (as listed in Table 4.1). The use of a high-performing solver called initially with a short time limit value discharges easy POs. It does this without incurring the cost of feature extraction and solver rank prediction. This heuristic is implemented successfully in the SATzilla [115] portfolio solver for SAT instances where it is termed "pre-solving". SATzilla has inspired the use of pre-solving in portfolio solvers for constraint / optimisation problems such as sunny-cp [6]. This heuristic improves Where4's performance and reduces its reliance on the underlying random forest prediction model. The process described in Algorithm 1 for obtaining results from solver rankings needs to be modified to describe Where4's operation: the following algorithm (Alg. 2) only performs feature extraction if the initial solver does not solve the input program within one second.

```
Input: P, a Why3 program;
R, a static ranking of solvers for pre-proving;
φ, a timeout value
Output: ⟨A, T⟩ where
A = the best answer from the solvers;
T = the cumulative time taken to return A
begin
    /* Highest ranking solver installed locally                */
    S ← BestInstalled(R)
    /* Call solver S on Why3 program P with a
       timeout of 1 second                                     */
    ⟨A, T⟩ ← Call(P, S, 1)
    if A ∈ {Valid, Invalid} then
        return ⟨A, T⟩
    end
    /* extract feature vector F from program P                 */
    F ← ExtractFeatures(P)
    /* R is now based on program features                      */
    R ← PredictRanking(F)
    while A ∉ {Valid, Invalid} ∧ R ≠ ∅ do
        S ← BestInstalled(R)
        /* Call solver S on Why3 program P with a
           timeout of φ seconds                                */
        ⟨A_S, T_S⟩ ← Call(P, S, φ)
        /* add time T_S to the cumulative runtime              */
        T ← T + T_S
        if A_S > A then
            /* answer A_S is better than the current
               best answer                                     */
            A ← A_S
        end
        /* remove S from the set of solvers R                  */
        R ← R \ {S}
    end
    return ⟨A, T⟩
end
```

Algorithm 2: Returning an answer and runtime from a Why3 input program

TABLE 5.1: Finding the minimal number of trees

N Trees	Time	R^2	nDCG	MAE	Reg-error
10	15.09	0.24	0.48	2.12	41.10
20	14.99	0.27	0.48	2.12	40.39
30	15.03	0.28	0.47	2.12	40.61
40	15.02	0.28	0.48	2.12	40.42
50	15.00	0.28	0.48	2.10	39.55
60	15.00	0.28	0.48	2.10	39.68
70	15.05	0.28	0.48	2.10	39.39
80	15.00	0.28	0.48	2.10	39.37
90	15.00	0.28	0.48	2.10	39.35
100	14.99	0.28	0.48	2.09	39.63

5.1 Finding the minimal number of trees

As previously discussed, the Random Forest algorithm operates by finding a prediction from each of its constituent decision trees and averaging their results. The generalisation of these results is improved by limiting the depth of these trees. The heuristic we used to ensure that no prediction was associated with less than five training instances was introduced in the previous chapter.

The Sci-Kit Learn implementation of the Random Forest algorithm uses 100 trees. When implementing our own version of the algorithm, the efficiency of deriving these predictions has to be taken into account. For that reason, we want to minimise the number of trees which have to be traversed.

As Table 5.1 shows, the prediction accuracy of the model does not deteriorate significantly as the number of trees decreases. We decided to use 50 trees for the final OCaml implementation: it can be seen that this number represents the point where the number of trees and the model's performance reaches stability. No significant increases in terms of Mean Average Error or Regression Error correspond to the use of more trees beyond this point.

5.2 Encoding the random forest

We train the Random Forest model described in Sec. 4.5.7 (and refined in the previous section) on the entire training set. The Sci-kit Learn library allows the constituent decision trees to be extracted and inspected. We print the forest as a JSON array of trees using the data model shown in Fig. 5.1.

This JSON schema is designed to be human-readable so that users can define their own simple trees in order to experiment with the effect a particular `feature` may have. The `index` attribute is a unique identifier used

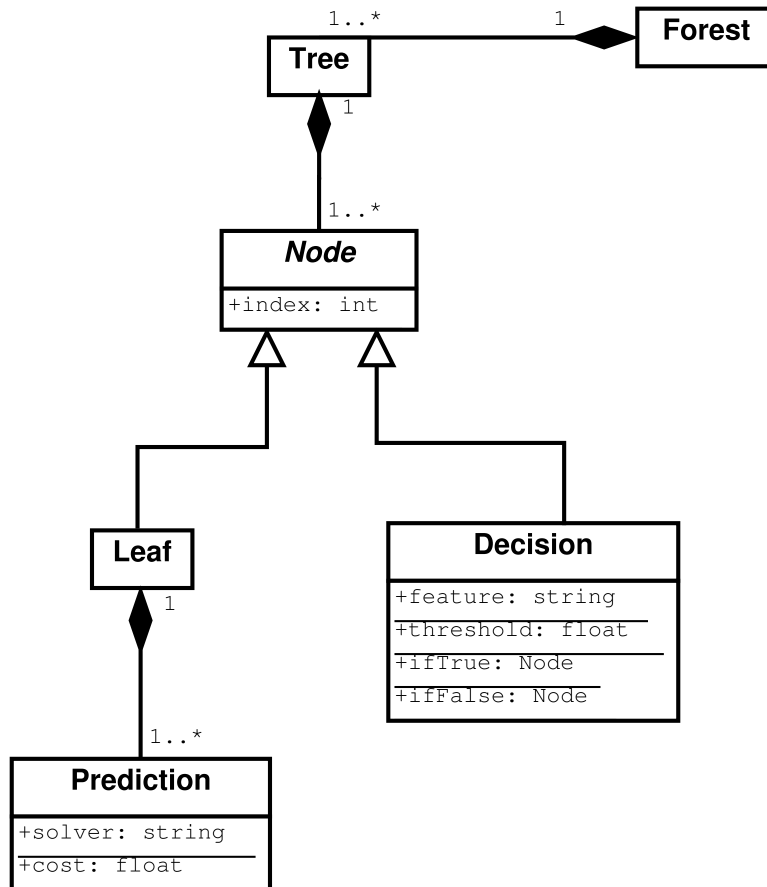

FIGURE 5.1: Data design for JSON encoding

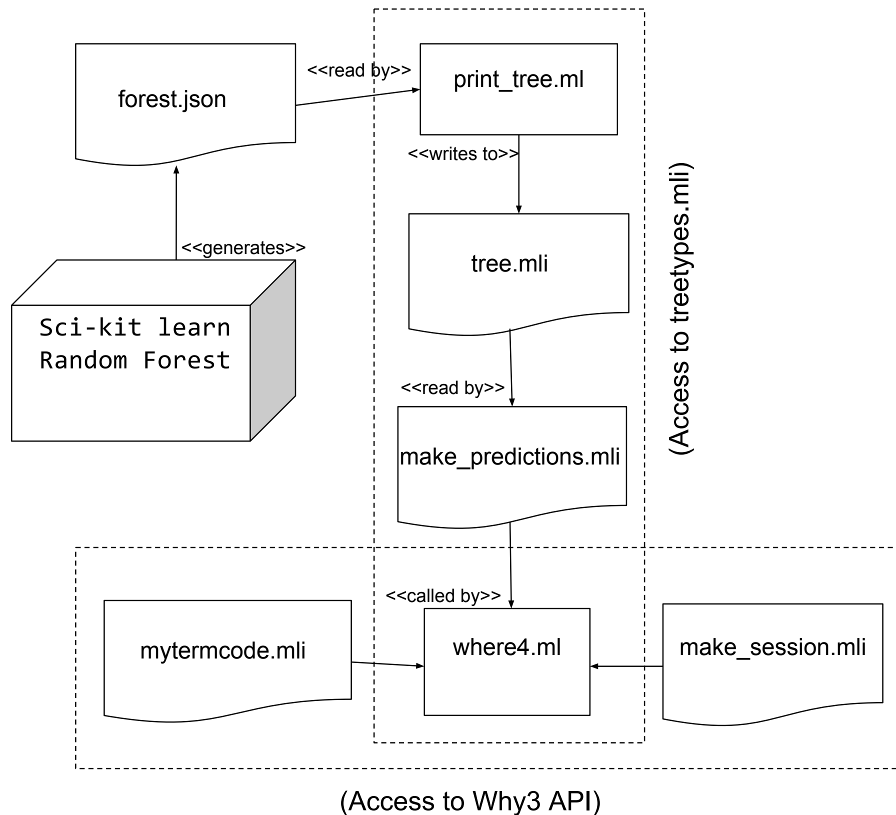

FIGURE 5.2: The organisation of components in Where4's design

when traversing the tree: if the value for `feature` is less than or equal to `threshold`, the current focus moves to the `Node` with the value of `ifTrue`, or to `ifFalse` if it is greater. This process continues until a `Leaf` node is encountered and each solver's cost `Prediction` is returned.

Fig. 5.2 shows Where4's design in terms of OCaml modules. The functions exposed by the interface files (`*.mli`) are listed in Appendix A. Upon installation of Where4, `forest.json` is read in by `print_tree.ml` and an equivalent OCaml array is generated and written to the file `tree.ml`. When `make_predictions.ml` is provided with a vector of program features by `where4.ml`, the forest is traversed in the manner outlined above. Making the forest available as an OCaml data structure (through the `tree.mli` interface file) is more efficient than reading in the JSON file at each execution of Where4. Of course, if `forest.json` changes, Where4 needs to be re-installed for these changes to have effect (more information about installing Where4 is given in Appendix C).

5.3 Extracting features

The previous section described Where4's use of a tree data structure to predict solver performance given a vector of program features. A tree is also used to derive these feature vectors from Why3 programs: the Why3 API is used to traverse the abstract syntax tree (AST) for each PO. As previously discussed in Sec. 3.2, we use a process similar to that used internally by Why3 to extract "goal shapes" [27]. The Why3 OCaml module used to perform this task is named `termcode.ml` and Where4's `mytermcode.ml` is based upon the same process.

A hash table is maintained while recursively traversing the AST: at each node visited the count of the corresponding feature is incremented in the hash table. Any other relevant information (such as function arity, for example) is also recorded before the rest of the tree is traversed.

5.4 Integration with Why3

The `mytermcode.mli`, `where4.ml` and `make_session.mli` files are shown in Fig. 5.2 as having access to the Why3 API. We have already described how `mytermcode.ml` uses the AST to extract the feature vector. Other important information required by Why3 is gathered by the `make_session.ml` module. For example, this module reads the Why3 configuration file to determine which supported SMT solvers are installed locally. `make_session.ml` loads drivers for these solvers and creates a proof session in the current directory. This module reads and type-checks the input file; returning its abstract representation. These actions are performed using a method described in the Why3 manual[1] [28].

`where4.ml` parses command line options (see Appendix B) and prints results to the console. The process it uses to schedule solvers is summarised by Algorithm 2. Where4 is available as a stand-alone command-line tool which accepts files written using the WhyML front-end or the Why IVL. Such a tool is useful as an initial solving step for large batches of files, but it cannot take full advantage of the Why3 system. By following the "zero click, zero maintenance, zero overhead" philosophy, we wanted Where4's functionality to integrate fully with the Why3 user's normal workflow. To take advantage of the full range of transformations and parsers associated with the Why3 system, we decided to make the tool available to be accessed as an individual backend solver.

The imitation of an orthodox SMT solver requires a number of modifications and extra files as follows:

[1]Specifically `http://why3.lri.fr/doc-0.87.2/manual005.html#sec28` (last accessed 16/10/16)

- An entry to the `provers-detection-data.conf` configuration file must be made for Why3 to recognise Where4 as a supported SMT solver. Where4's entry lists the command needed to invoke the binary, specifies which version of the tool is supported, and where to find the corresponding driver file.
- The driver file must contain regular expressions to parse Where4's output, making it comprehensible to Why3 (defining the characteristics of a *Valid* output, for example). The driver must also list the printer used by Why3 for intermediate files, as well as any transformations which need to be performed to conform to the solver's input language. Where4 uses the standard Why3 printer and its list of transformations.
- The Why3 API requires that supplied paths to input files are relative to the current directory. When called via the driver, however, a temporary file is written by the printer and an absolute path is specified. Where4 needs to convert this to a relative path in order to read the input file.

These modifications allow Where4 to be recognised as an orthodox solver which can be used by Why3 through the IDE, on the command-line, or through the OCaml API – similar to any other supported ATP or SMT tool.

5.5 Summary

This chapter has presented an overview of the method we used to implement the Where4 tool using the Why3 OCaml API. In total, the six statically-written OCaml modules (i.e. not including `tree.ml`) contain 712 lines of code. We used OCaml version 4.02.3 for their compilation locally. The only library we used (in addition to version 0.87.1 of the Why3 API) is Yojson[2] to help with parsing the `forest.json` file. The data disk included with this thesis contains all of the source code for Where4. The files `install.sh` and `readme.md` contain useful information about the compilation of these modules and how to satisfy their few dependencies. The same source files are available online at https://github.com/ahealy19/where4.

[2]version 1.2.1 http://mjambon.com/yojson.html

Chapter 6

Evaluating Where4 on Test Data

This chapter will evaluate Where4's portfolio algorithm on the held-back test data. The randomly-selected test set represents 25% of the entire number of POs and consists of 32 WhyML files, 77 theories and 263 goals (see Sec. 4.3.2). In addition to evaluating Where4 's *predictions*, the OCaml implementation (detailed in the previous chapter) will be discussed in terms of its efficiency. We also present the results of our prediction algorithm on the training data when no pre-solver is used (i.e. Algorithm 1 is used). We do this to evaluate the prediction model in isolation and compare Where4's performance with and without the pre-solving heuristic. We perform our evaluation guided by three Evaluation Questions:

EQ1: **How does Where4 perform in comparison to the eight SMT solvers?**
The importance of this question is obvious: the success of Where4 depends on its improvement over the status quo. In the case of discharging Why3 POs, the status quo is represented as the use of a single solver.

EQ2: **How does Where4 perform in comparison to the three theoretical strategies?**
The theoretical strategies introduced in Sec. 4.4 provide a fairer basis for comparison than a single solver by taking multiple solver calls per PO into consideration. As a reminder for the reader, the Best Ranking always chooses solvers in the order of ascending cost, Random Ranking is the average result of running *every* possible permutation of the eight solvers, and Worst Ranking is the inverse of Best Ranking: it is the ranking of solvers in order of *descending* cost.

EQ3: **What is the time overhead of using Where4 to prove Why3 goals?**
The feature extraction and solver scheduling processes incur a time cost. This evaluation criterion measures whether this cost represents a significant proportion of Where4 's overall solving time.

TABLE 6.1: Number of files, theories and goals proved by each strategy and individual solver. The percentage this represents of the total 32 files, 77 theories, 263 goals, and the average time in seconds, are also shown.

	File			**Theory**			**Goal**		
	# proved	% proved	Avg time	# proved	% proved	Avg time	# proved	% proved	Avg time
Where4 (PS)	11	34.4%	1.39	44	57.1%	0.99	203	77.2%	1.98
Where4 (no PS)			1.99			1.31			2.32
Best Rank.			0.25			0.28			0.37
Random Rank.			4.19			4.02			5.70
Worst Rank.			14.71			13.58			18.35
Alt-Ergo-0.95.2	8	25.0%	0.78	37	48.1%	0.26	164	62.4%	0.34
Alt-Ergo-1.01	10	31.3%	1.07	39	50.6%	0.26	177	67.3%	0.33
CVC3	5	15.6%	0.39	36	46.8%	0.21	167	63.5%	0.38
CVC4	4	12.5%	0.56	32	41.6%	0.21	147	55.9%	0.35
veriT	2	6.3%	0.12	24	31.2%	0.12	100	38.0%	0.27
Yices	4	12.5%	0.32	32	41.6%	0.15	113	43.0%	0.18
Z3-4.3.2	6	18.8%	0.46	31	40.3%	0.20	145	55.1%	0.37
Z3-4.4.1	6	18.8%	0.56	31	40.3%	0.23	145	55.1%	0.38

This chapter answers each Evaluation Question in turn. Threats to the validity of our study are discussed in Sec. 6.4.

6.1 EQ1: How does Where4 perform in comparison to the eight SMT solvers?

When each solver in Where4's ranking sequence is run on each goal, the maximum amount of files, theories and goals are provable. As previously mentioned in Sec. 4.4.2 and as Table 6.1 shows, the difference between Where4 and the set of reference theoretical strategies (Best Ranking, Random Ranking, and Worst Ranking) is the amount of time taken to return the *Valid/Invalid* result. Compared to the eight SMT solvers, the biggest increase is on individual goals: Where4 can prove 203 goals, which is 26 (9.9%) more goals than the next best single SMT solver, Alt-Ergo-1.01.

As is shown by Table 6.1, the average time taken for Best Ranking to return an answer of *Valid* is not necessarily less than that of an individual solver. As Best Ranking can return *Valid* for all provable POs – more than any individual solver – it is the number of *Valid* answers, rather than any inefficiency, that is responsible for this slightly slower average time.

The average time Where4 takes to prove these goals (using Algorithm 2 – denoted Where4 (PS) in Table 6.1) is significantly better than the Random Ranking strategy. Without a pre-solver, Where4 suffers slightly (as denoted

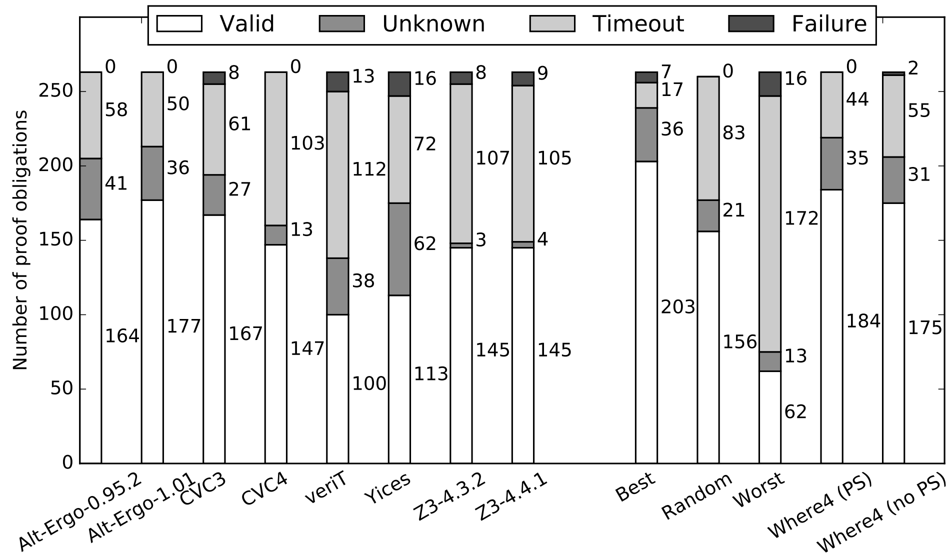

FIGURE 6.1: The relative amount of Valid/Unknown/Timeout/Failure answers from the eight SMT solvers, Where4 (PS where pre-solver = Alt-Ergo-1.01), Where4 with no Pre-Solver, and three theoretical strategies on the 263 test POs (with a timeout limit of ten seconds).

by the results for Where4 (no PS)). Both algorithms, however, lag behind the time recorded by the Best Ranking strategy.

In comparison to the eight SMT solvers, the average time taken by Where4 to prove each of the 203 goals is high. This tells us that Where4 can perform badly with goals which are not provable by many SMT solvers: expensive *Timeout* results are chosen before the *Valid* result is eventually returned. In the worst case, Where4 may try all eight solvers in sequence, timing out for each solver, whereas each individual solver does this just once. Thus, while having access to more solvers can allow more goals to be proved (if there are goals uniquely-provable by the solvers such as those identified in Table 4.2), there is also a time penalty associated with portfolio-based solvers in these circumstances. This issue has previously been identified by Amadini et al. in their studies on portfolio solvers for constraint programming [4] and constraint optimisation [5] where portfolio performance was found to degrade as the number of constituent solvers increased.

The multiple timeout issue raises the question of whether it is fair to compare Where4 to individual solvers. Any ranking strategy will be able to prove the maximum number of files, theories and goals, but unless the best solver is consistently placed high in the ranking, it could take a significantly longer time to do so than even the worst-performing individual solver.

We remind the reader of the Choose Single solver introduced in Sec. 4.1. Choose Single is the *best single solver* as chosen on a per-goal basis. It

provided a motivation for the use of portfolio-solving on the Why3 platform by proving the maximum number of goals in the shortest amount of time. We mentioned that Choose Single is equivalent to choosing the top-ranking solver from Best Ranking and stopping. We return to this concept in Fig. 6.1 which is similar to Fig. 3.6 in that it shows the relative amount of *Valid/Unknown/Timeout/Failure* answers from the eight SMT solvers. Also shown (on the right) are results obtainable by using the top solver (only) with the three ranking strategies (where Best Ranking $\equiv$ Choose Single) and the Where4 predicted ranking (after pre-solving with Alt-Ergo version 1.01 – Where4 (PS) – and without: Where4 (no PS)).

```
Input: P, a Why3 program;
R, a static ranking of solvers for pre-proving;
φ, a timeout value;
μ, the cost threshold
Output: ⟨A, T⟩ where
A = the best answer from the solvers;
T = the cumulative time taken to return A
begin
    /* pre-solving                                          */
    S ← BestInstalled(R)
    ⟨A, T⟩ ← Call(P, S, 1)
    if A ∈ {Valid, Invalid} then
        return ⟨A, T⟩
    end
    F ← ExtractFeatures(P)
    R ← PredictRanking(F)
    /* the predicted cost of S is an additional
       stopping condition                                   */
    while A ∉ {Valid, Invalid} ∧ R ≠ ∅ ∧ Cost(S) ≤ μ do
        S ← BestInstalled(R)
        ⟨A_S, T_S⟩ ← Call(P, S, φ)
        T ← T + T_S
        if A_S > A then
            A ← A_S
        end
        /* remove S from the set of solvers R               */
        R ← R \ {S}
    end
    return ⟨A, T⟩
end
```

Algorithm 3: Returning answers and runtimes from Where4 solver rankings using a cost threshold: A minor modification to Alg. 2 with an additional stopping condition in the **while** loop

The 62 *Valid* answers returned by the top solver from the Worst Ranking (i.e. the worst solver) represent the trivial POs solvable by all eight solvers. Likewise, the 60 goals for which Best Ranking did not return a *Valid* or *Invalid* answer could not be proved by any solver.

The results show that limiting the portfolio solver to just using the best predicted individual solver eliminates the multiple time-out overhead yet reduces the number of goals provable by Where4. This number of goals – 184 – is still more than the best-performing individual SMT solver, Alt-Ergo version 1.01.

In an effort to compare Where4 to individual SMT solvers, Table 6.1 and Fig. 6.1 show results at two extremes of a spectrum: using all solvers available, and only using one. In the next subsection we describe a method to calibrate the use of Where4 by using the predicted cost of each solver.

6.1.1 Use of a cost threshold

To balance the time-taken-versus-goals-proved trade-off associated with the two approaches above, we introduce the notion of a *cost threshold* as another method of comparing Where4 to individual SMT solvers. Where4's use of a cost threshold constitutes a minor adjustment to Alg. 2 and is detailed in Alg. 3. After pre-solving, solvers with a predicted cost above this threshold are not called. If every solver's cost is predicted to be above the threshold μ, the pre-solver's result is returned.

We determine the appropriate value for this threshold by first splitting the training data into model training and validation sets. The model training set used for this step represents 90% of the total training set (or 706 POs), while the validation set is made up of 79 POs. We train the Random Forest predictor (with pre-solving) before simulating the effect of an increasing cost threshold using the validation set.

Fig. 6.2a and 6.2b show the effect of varying this threshold when solving POs in the test set. The top plot (*a*) shows a comparison of the average time taken for *any* answer to be returned (not necessarily *Valid* / *Invalid*). The amount of time taken by Where4 often depends on the number of solvers called. The number of solvers called depends on the cost threshold given to Where4. This is particularly true in the case where a pre-solver is not used. The solid black/red lines of Fig. 6.2a, shows the increase in average time taken by Where4 (with / without pre-solving) to return a response as the cost threshold increases. Fig. 6.2b shows the number of *Valid* / *Invalid* responses returned by each individual SMT solver as compared to Where4 with a range of threshold values. As the corresponding results for each individual solver are unaffected by the threshold parameter, they are represented by horizontal line segments intersecting with the Where4 data in Fig. 6.2a.

We note that even when given a threshold value of zero, 46 POs are proven. The difference with the solid red line's more gradual increase makes it obvious that the pre-solving routine is responsible for these results. In our case, Alt-Ergo-1.01 can prove 46 POs given a time limit of one

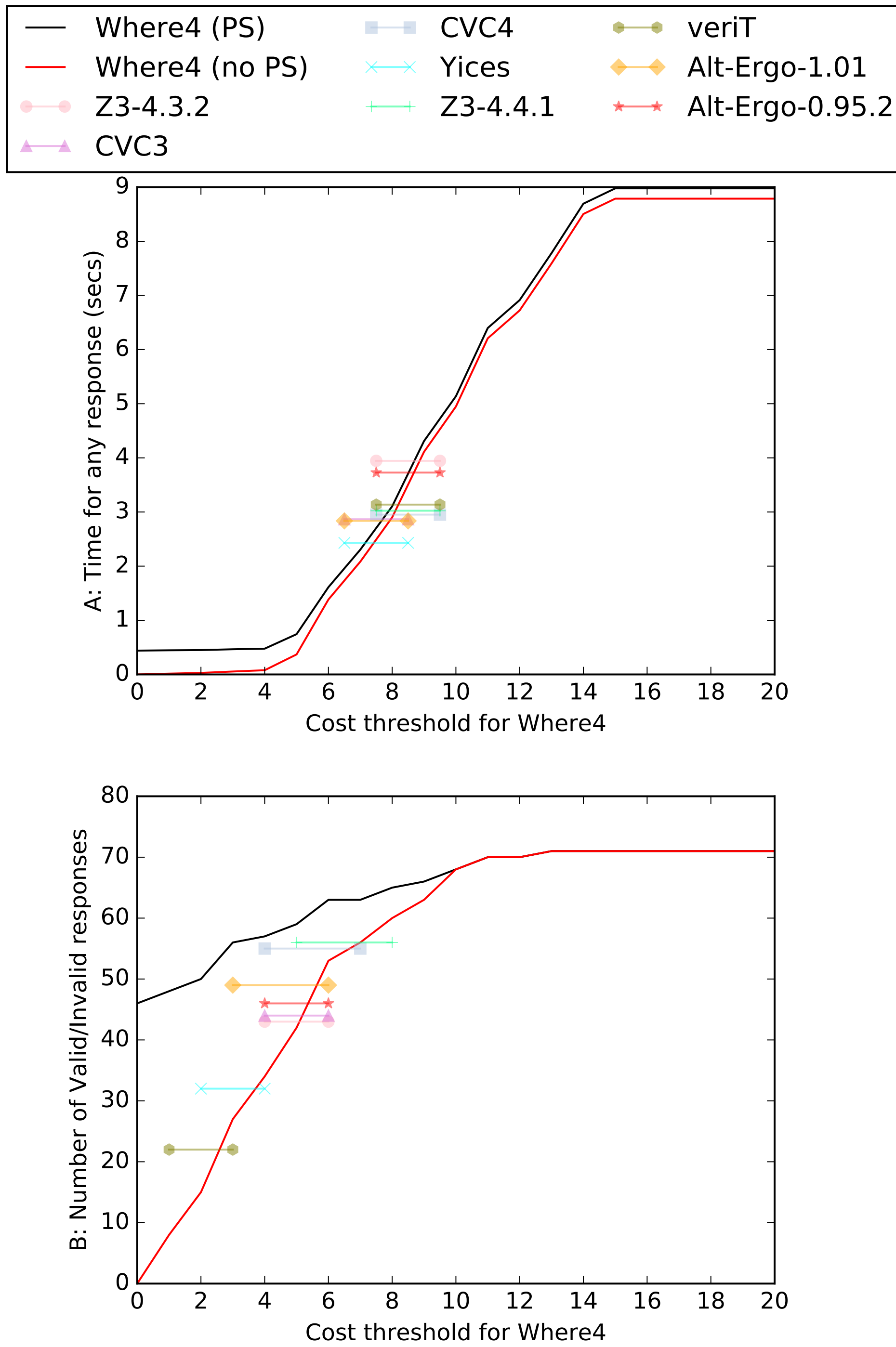

FIGURE 6.2: Determining the appropriate value for a cost threshold (using a validation dataset). (*a: top plot*) The average time taken for Where4 to return an answer compared to eight SMT solvers. (*b: bottom plot*) The number of Valid/Invalid answers returned by Where4 compared to eight SMT solvers.

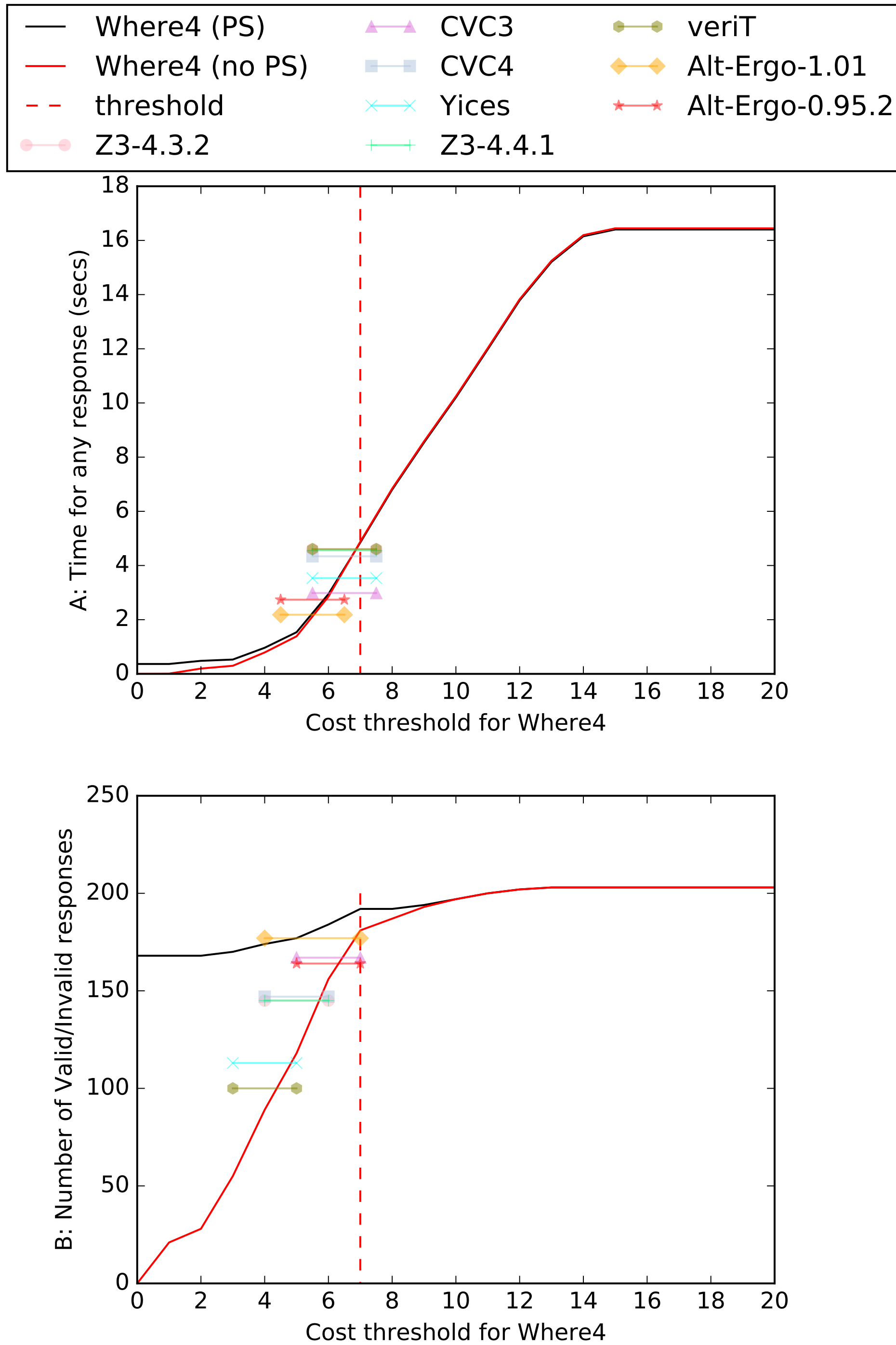

FIGURE 6.3: The effect of using a cost threshold. (*a: top plot*) The average time taken for Where4 to return an answer compared to eight SMT solvers. (*b: bottom plot*) The number of Valid/Invalid answers returned by Where4 compared to eight SMT solvers.

second and the solver cost is never used. As in Fig. 6.2a, the results for each individual solver are unaffected by the threshold parameter.

By inspecting these results, we find that a threshold value of seven gives good results. At this threshold, the use of a pre-solver results in far more *Valid/Invalid* responses being returned than the best-performing individual solver (Z3-4.4.1) and they are returned in a faster time (on average) than the fastest-returning solver (Yices). Even without pre-solving (the solid red line), about the same number of *Valid/Invalid* responses are returned as by Z3-4-4-1 and in a slightly faster time than the pre-solving version.

As the dashed red line in Fig. 6.3 shows, a threshold value of seven also performs well on the test data. The results for Where4 using a pre-solver are shown in numerical form in Table 6.2. When Where4 is given a cost threshold of five, it can prove the same number of POs as the best-performing solver – Alt-Ergo-1.01. By referring to Fig. 6.3a, we see that at the same cost threshold, it takes a shorter time to return a response, on average, than the fastest SMT solver (which is also Alt-Ergo-1.01). If the cost threshold is increased to seven, significantly more POs can be proven. The average time taken to return a response is approximately equal to that of the four slowest individual solvers on the test data: CVC4, veriT, and both versions of Z3.

EQ1 Answer: The cost threshold greatly improves Where4's performance in comparison to the individual SMT solvers. The performance penalties associated with portfolio solvers can be mitigated by defining a cut-off point and trusting that solvers with a predicted cost greater than this value do not need to be called. We found this point to be about seven for the POs in the Why3 example dataset. Giving Where4 a threshold value less than seven may result in significantly worse results without the use of a pre-solver. The value to choose as a threshold may not be obvious in real-world scenarios with unseen results, however.

6.2 EQ2: How does Where4 perform in comparison to the three theoretical strategies?

Fig. 6.4 compares the time taken for Where4 (with and without the use of a Pre-Solver) and the three ranking strategies to return *Valid* answers for the 263 POs in the test set. This experiment is equivalent to running each strategy on all 263 POs in parallel and measuring the time taken for each strategy to return a total of 203 *Valid* answers. As described in Sec. 4.4, Best Ranking and Worst Ranking use Alg. 1 to return their time taken. Random Ranking's time measurement is based on the average of 40,320 (i.e. eight

TABLE 6.2: The effect of using a cost threshold. The average time taken for Where4 (with pre-solver) to return an answer compared and the number of Valid/Invalid answers. Same data as Fig. 6.3

THRESHOLD	**0**	**1**	**2**	**3**	**4**	**5**
Avg. Time	0.37	0.37	0.48	0.53	0.98	1.77
Num. Proved	168	168	168	169	173	177
THRESHOLD	**6**	**7**	**8**	**9**	**10**	**11**
Avg. Time	2.72	4.59	6.55	8.74	10.31	11.89
Num. Proved	183	192	192	195	197	199
THRESHOLD	**12**	**13**	**14**	**15**	**16**	**17**
Avg. Time	14.16	15.24	16.06	16.35	16.35	16.35
Num. Proved	202	203	203	203	203	203

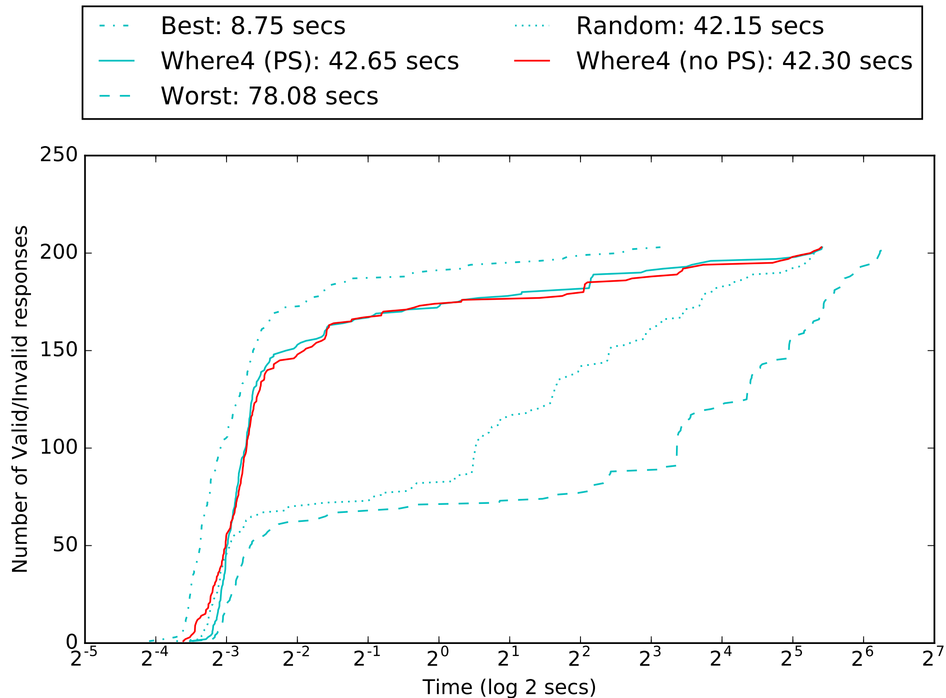

FIGURE 6.4: The time taken by each theoretical strategy and Where4 to return all *Valid/Invalid* answers in the test dataset of 263 goals

factorial, the number of individual rankings) uses of Alg. 1. Where4 (PS)'s times are derived from the use of Alg. 2 with a pre-solver of Alt-Ergo-1.01.

Although both Where4 implementations and Random Ranking finish at approximately the same time, Where4 is significantly faster for returning *Valid/Invalid* answers. Both Where4's solid lines are more closely correlated to Best Ranking's rate of success than the erratic rate of the Random Ranking strategy. Surprisingly, the implementation of Where4 which does not use a pre-solver (Fig. 6.4's solid *red* line), finds slightly more *Valid/Invalid* answers at the lower end of the time scale than the pre-solver. We assert that this is a validation of the prediction algorithm's effectiveness: amongst provable POs, it often chooses a solver faster than the pre-solver. Again, we find that Where4 struggles to prove a small number of POs. This trait can be seen in the amount of time needed for either Where4 implementation to return all *Valid/Invalid* responses (42.65/42.30 seconds) compared to that required by the Best Ranking strategy (just 8.75 seconds). Best Ranking's excellent time result shows the capability of a perfect-scoring learning strategy.

EQ2 Answer: At almost any point in time from zero seconds to 42.65 seconds, the number of *Valid/Invalid* answers returned by Where4 is greater than the number returned by Random Ranking. This shows that our prediction model is a better choice than selecting a sequence of SMT solvers at random, without any regard for program features or solver capability. Where4's performance on all but the hardest POs is encouraging. Although Where4 cannot compete with Best Ranking yet, the performance of this theoretical strategy is motivation to further improve Where4 in the future. Over the entire test dataset (i.e. both provable and non-provable PO instances), there was no significant time time advantage associated with using a pre-solver.

6.3 EQ3: What is the time overhead of using Where4 to prove Why3 goals?

The timings for Where4 in all plots and tables in this chapter are based solely on the performance of the constituent solvers (the measurement of which is discussed in Sec. 3.3). They do not measure the time it takes for the OCaml binary to extract the static metrics, traverse the decision trees and predict the ranking. We have found that this adds (on average) 0.46 seconds to the time Where4 takes to return a result for each of the files in the test set. On a per goal basis, this is equivalent to an increase in 0.056 seconds. This overhead is only applied to goals for which pre-solving is unsuccessful. For the test set, pre-solving eliminated this overhead for 168

out of 263 POs (see Table 6.2: Where4's number of *Valid*/*Invalid* responses when the cost threshold is zero).

The imitation of an orthodox solver to interact with Why3 is more costly: this is due to Why3 printing each goal as a temporary file to be read in by the solver individually (see Sec. 5.4). As Where4 uses the abstract, internal representation of the program, the printing of each goal to the Why format is unnecessary. The Why3 driver mechanism makes this step unavoidable for any supported solver including, of course, the Where4 "imitation solver".

Another issue with calling Where4 through Why3 is that of applying Why3's timeout value to a sequence of solver calls. For example, if a user gives Why3 a timeout of five seconds for Where4 to prove a PO and the first solver called by Where4 goes over that limit, the potentially useful answer returned by the second solver in the sequence would not be returned to the user.

EQ3 Answer: While pre-solving is an important heuristic for avoiding the overhead of feature extraction and rank prediction, these processes must be optimised for Where4 to be practical as a portfolio solver called from Why3. Future work will look at a portfolio-solving Why3 plugin similar to Where4. Tighter integration with the Why3 system can be expected to improve the efficiency of such a tool. In its current form, Where4 is more suited to an initial proving step performed using the stand-alone command line tool.

6.4 Threats to Validity

In this section we discuss the threats to the validity of the evaluation presented in this chapter. We categorise threats as either *internal* or *external*. Internal threats refer to influences that can affect the response variable without the researcher's knowledge and threaten the conclusions reached about the *cause* of the experimental results [111]. Threats to external validity are conditions that limit the generalisability and reproducibility of an experiment.

6.4.1 Internal

The main threat to our work's internal validity is selection bias. All of our training and test samples are taken from the same source. We took care to split the data for training and testing purposes on a *per file* basis, as we discussed in Sec. 4.3.2. This ensured that Where4 was not trained on a goal belonging to the same theory or file as any goal used for testing.

The results of running the solvers on our dataset are imbalanced: (i) there were far more *Valid* responses than any other response and (ii) no goal

in our dataset returned an answer of *Invalid* on any of the eight solvers. Issue (i) could be remedied through the use of resampling. The technique either adds copies of instances of under-represented classes (*over-sampling*) or remove instances of POs returning *Valid* responses (*undersampling*). A larger dataset of POs would help with resampling to ensure a representative balance of responses is reflected in the test and training set. Issue (ii) is a more serious problem as Where4 would not be able to recognize such a goal in real-world use.

Use of an independent dataset is likely to influence the performance of the solvers. Alt-Ergo was designed for use with the Why3 platform – its input language is a previous version of the Why logic language. It is natural that the developers of the Why3 examples would write programs which Alt-Ergo in particular would be able to prove. Due to the syntactic similarities in input format and logical similarities such as support for type polymorphism, it is likely that Alt-Ergo would perform well with any Why3 dataset. We would hope, however, that the gulf between it and other solvers would narrow.

There may be confounding effects in a solver's results that are not related to the independent variables we used (Sec. 3.2). We were limited in the tools available to extract features from the domain-specific Why logic language (in contrast to related work on model checkers which use the general-purpose C language [49, 108] – as previously discussed in Sec. 2.3.2). We made the decision to keep the choice of independent variables simple in order to increase generalisability to other formalisms such as Microsoft's Boogie [12] intermediate language.

6.4.2 External

The generalisability of our results is limited by the fact that all dependent variables were measured on a single machine. All data collection was conducted on a single 64-bit machine running Ubuntu 14.04 with a dual-core Intel i5-4250U CPU and 16GB of RAM. We believe that the number of each response for each solver would not vary dramatically on a different machine of similar specifications. By inspecting the results when each solver was given a timeout of 60 seconds (Fig. 3.7), the rate of increase for *Valid/Invalid* results was much lower than that of *Unknown/Failure* results. The former set of results are more important when computing the cost value for each solver-goal pair.

Timings of individual goals are likely to vary widely (even across independent executions on the same machine). It is our assumption that although the actual timed values would be quite different on any other machine, the *ranking* of their timings would stay relatively stable.

A "typical" software development scenario might involve a user verifying a single file with a small number of resultant goals: certainly much smaller than the size of our test set (263 goals). In such a setting, the productivity gains associated with using Where4 would be minor. Where4 is more suited therefore to large-scale software verification.

6.5 Discussion

By considering the answers to our three Evaluation Questions, we can make assertions about the success of Where4. The answer to **EC1**, Where4's performance in comparison to individual SMT solvers, is positive. A small improvement in *Valid/Invalid* responses results from using only the top-ranked solver, while a much bigger increase can be seen by making the full ranking of solvers available for use. The time penalty associated with calling a number of solvers on an un-provable PO is mitigated by the use of a *cost threshold*. Judicious use of this threshold value can balance the time-taken-versus-goals-proved trade-off: in our test set of 263 POs, using a threshold value of seven results in 192 *Valid* responses – an increase of fifteen over the single best solver – in a reasonable average time per PO (both *Valid* and otherwise) of 4.59 seconds.

There is also cause for optimism in Where4's performance as compared to the three theoretical ranking strategies – the subject of Evaluation Question 2. All but the most stubborn of *Valid* answers are returned in a time far better than Random Ranking. We take this random strategy as representing the behaviour of the non-expert Why3 user who does not have a preference amongst the variety of supported SMT solvers. For this user, Where4 could be a valuable tool in the efficient initial verification of POs through the Why3 system.

In terms of time overhead – the concern of EQ3 – our results are less favourable, particularly when Where4 is used as an integrated part of the Why3 toolchain. The costly printing and parsing of POs slows Where4 beyond the time overhead associated with feature extraction and prediction. At present, due to the diversity of languages and input formats in use for SV (see Sec. 2.1), this is an unavoidable pre-processing step enforced by Why3 (and is indeed one of the Why3 system's major advantages).

Overall, we believe that the results for two out of three Evaluation Questions are encouraging and suggest a number of directions for future work to improve Where4.

Chapter 7

Conclusion

This thesis has presented a strategy to choose appropriate SMT solvers based on the syntactic features of Why3 POs. Our final solution, Where4, was implemented after a careful consideration of a number of datasets (Sec. 2.1.1), SMT-solving tools (Sec. 3.1), learning tasks (Sec 4.2) and learning algorithms (Sec. 4.3).

The Where4 tool is a random-forest multi-output regressor with a number of optimisation heuristics. Users without any knowledge of SMT solvers (i.e. those choosing solvers at random) can prove a greater number of goals in less time by delegating to Where4. Although the prediction accuracy of Where4 is disappointing on some instances of the test set, we maintain that Where4 represents a positive first step in developing the portfolio solver that simplifies discharging POs for non-expert Why3 users.

We believe that the Why3 platform has great potential for machine-learning based portfolio-solving due to its unique approach to interfacing with disparate ATP, SMT and ITP tools. We are encouraged by the performance of a theoretical Best Ranking strategy. The convenience that a tool implementing such a strategy would give Why3 users has the potential to make deductive software verification more approachable to the wider software engineering community. This ultimate goal provides a motivation for improving Where4 through future work.

7.1 Future Work

The number of potential directions for this work is large. An obvious and immediate first step could be the use of a larger and more generic dataset for training and testing purposes. The two viable alternatives to the Why3 example dataset discussed in Sec. 3.1.1 – the TPTP library [106] and the BWARE dataset [48] – could be investigated to this end.

The application of an initial splitting transformation (to simplify each PO) would also be relatively simple to implement. This would also create an entirely different, and potentially much larger, dataset. For example, applying the `split_goal_wp` transformation to each PO in the Why3 example dataset would increase its size from 1048 to 7489 POs. Of course, any

changes to the dataset would require repeating the costly measurement of dependent variables (Sec. 3.3).

An interesting direction for this work could be the identification of the simplifying transformations (both splitting and non-splitting) which would need to be applied by Why3 in order for a PO to be tractable for an SMT solver. The tool resulting from this multi-class classification task would be quite different to the one implemented as Where4 but potentially very useful. It would complement the existing Where4 tool in its goal to assist non-expert Why3 users in discharging proof obligations through automation.

Applying multiple solvers in parallel could potentially alleviate some of the time penalties associated with Where4 and portfolio solvers in general (identified in Sec. 6.1). The effectiveness of such an approach has been demonstrated by the `ppfolio` (Parallel PortFOLIO) SAT tool which won eleven medals at the SAT 2011 competition [86]. All SMT solvers could either be scheduled to run in parallel (thereby eliminating the need for feature extraction and prediction) or alternatively only the solvers with the lowest predicted cost could be run. There is a trade-off in the allocation of computational resources associated with these two approaches: the time and CPU cycles needed for feature extraction and solver prediction must be balanced with those consumed by a number of low-performing solvers.

One of the limitations of this work is lack of comparison of features for extraction. Future would could make a more thorough investigation of how semantic properties of programs (eg. the associativity of the operators found in the program, constructions indicating inductively defined types, etc.) affect accurate predictions for each learning algorithm.

As mentioned in Sec. 6.3 in response to Evaluation Question 3, future work could investigate the re-implementation of Where4 as a Why3 plugin in order to optimise the interaction with Why3's internal data structures used for feature extraction. Aside from the implementation of the Where4 tool itself, a *minimal benchmark suite* could be identified which could be used to train the prediction model using new SMT solvers and theorem provers installed locally.

Appendix A

Ocaml Interfaces

treetypes.mli

```
type prediction = (string * float)

type decision = (string * float)

type tree_node =
  | Node of (decision * int * int)
  | Leaf of prediction list

type decision_tree = tree_node array

type forest = decision_tree list

type tree_or_forest =
  | Tree of decision_tree
  | Forest of forest

(* built with: ocamlfind ocamlc -c treetypes.mli *)
```

tree.mli

```
open Treetypes

val tree : tree_or_forest
```

mytermcode.mli

```
(*  based on 'shape' computation function:
  val t_shape_task: ?version:int -> Task.task -> shape
  from Why3: src/session/termcode.mli *)

open Why3

(*  the function that traverses the AST of a Why3 Task
    .
  A hash table containing the number/value of each
    feature
  is returned.

  This is encoded as feature (string) -> value (float)
 *)
val t_shape_num_map: Task.task -> (string, float)
    Hashtbl.t
```

get_predictions.mli

```
open Treetypes

(*  given the feature vector returned from mytermcode.
    ml,
  return the corresponding leaf node *)
val get_predictions: (string, float) Hashtbl.t ->
    tree_node option

(* if no proving is to be done, predictions can be
    printed *)
val print_predictions: tree_node option -> unit

val sort_predictions : tree_node option -> prediction
    list

(* best solver installed locally, that is *)
val get_best : prediction list -> string option
```

make_session.mli

```
open Why3

(* takes the path to the .why/.mlw file, the path to
   the .why.conf file
  and the file format (why or whyml) as arguements  *)
val make_file : string -> string -> string -> (unit
   Session.file)

(*  by default, Why3 adds './' to any path supplied.
  However, when called by the driver, an absolute
  path is supplied - this converts the absolute path
  to a relative one *)
val make_relative : string -> string

(* which solvers (known to Where4) are installed
   locally? *)
val provermap : (string, (Whyconf.config_prover *
   Driver.driver) option)
           Hashtbl.t
```

Appendix B

Where4 command-line options

Much like Why3 itself, the Where4 tool is intended for use on Unix systems only. During the installation of Where4, the location of a binary called **`where4`** is added to the user's `PATH` environment variable. The following options are appended to calls to **`where4`** on the command-line:

`--help / -h`
Print a consise version of this Appendix to the console.

`--version`
Print Where4's version number. This command is used by Why3 to determine if a supported version of Where4 is installed.

`--list-provers / -l`
Print each SMT solver known to Where4 and `found` if Where4 has determined it is installed locally; `NOT found` otherwise.

The following two options, `predict` and `prove`, must be followed by `FILENAME`: a *relative* path to a `.why` (Why logic language) or `.mlw` (WhyML programming language) file.

`predict FILENAME`
Print the predicted ranking of solver utility for each PO in `FILENAME`. The ranking will consist of all 8 solvers known to Where4 whether they are installed locally or not.

`predict FILENAME`
Call the pre-solver, then each *installed* solver in the predicted ranking sequentially, for each PO in `FILENAME`.

If the Why3 configuration file has been moved to a location other than its default (i.e. `$HOME/.why3.conf`), the following option is necessary in order for `where4 prove` or `where4 predict` to function correctly:

`--config / -c CONFIGPATH`
Specify the path to Why3's `.why.conf` configuration file as `CONFIGPATH`. Use the default location (`$HOME/.why3.conf`) otherwise.

A number of optional parameters can be appended to the `prove FILENAME` command (their order is unimportant):

`--verbose`
Print the result of each prover call and the time it took. The default behaviour is to just print out the best result and the cumulative time (see Alg. 3).

`--why`
A special flag for use with the Why3 driver which tells Where4 to convert the given absolute filename to a relative path.

`--time / -tm TIME`
Override the default (5 seconds) timeout value with `TIME` number of seconds. `TIME` must be an integer; an error message will be printed otherwise.

`--threshold / -ts THRESH`
Use a *cost threshold* (see Sec. 6.1.1) to limit the number of solvers called based on their predicted ranking. We found in Chapter 6 that a `THRESH` value of 7 is optimum for the test set. `THRESH` must be parsable as a floating-point number; an error message will be printed otherwise.

B.1 The Where4 command given to Why3

The following is the command specified in `provers-detection-data.conf` to call Where4:

```
where4 prove %f --why -tm %t -ts 7
```

`%f` is the absolute path to the temporary file created by Why3

`--why` tells Where4 that the given path is absolute and requires conversion

`-tm %t` use the time limit %t specified by the Why3 user

`-ts 7` enforce a cost threshold of 7

Appendix C

Where4 installation options

The installation process for Where4 is defined by a shell script `install.sh`. This Appendix gives more details about the options available to the user when calling this file to compile the OCaml files and install the Where4 binary locally.

`--location / -l PATH`
By default, Where4 will be copied to the `/usr/local/bin/` directory. If the user wishes, this location can be overridden to be `PATH`. The given directory must be on the user's `$PATH` environment variable to be found by Why3.

`--why3name / -w PATH`
The location of the Why3 binary, if it is not on the user's `$PATH` can be supplied. Where4 is added to Why3's list of provers by calling `why3 config -detect-provers` at the end of the installation process.

`--prover-detection / -p PATH`
By default, Why3's `provers-detection-data.conf` file is assumed to be located in `/usr/local/share/why/`. The location of this file can be specified as `PATH`.

`--driver-location / -d PATH`
The location of the Why3 driver files. Where4 needs to know where to copy `where4.drv` so that Why3 will be able to find it.

`--reinstall / -r`
Delete the installed binary (i.e. execute `uninstall.sh`) and repeat the installation process (the Where4 entry in `provers-detection-data.conf` will not be deleted, however). This option can be combined with the above flags.

By default, Where4 assumes there is a JSON file called `forest.json` in the current directory which is to be used to construct the prediction model. The user can control this behaviour with the following flags. As a random forest is just an array of decision trees, a JSON file containing a single tree may be provided instead, if it is specified as such during installation.

--forest / -f PATH
Use the JSON file located at `PATH` to construct the prediction model. This file should define a Random Forest using the JSON schema defined in Sec. 5.2.

--tree / -t PATH
Use the JSON file located at `PATH` to construct the prediction model. This file should define a Decision Tree using the JSON schema defined in Sec. 5.2.

Bibliography

[1] ISO/IEC 19761:2011. *Software engineering — COSMIC: a functional size measurement method*. 2011.

[2] Behzad Akbarpour and Lawrence C. Paulson. "MetiTarski: An Automatic Prover for the Elementary Functions". In: *Intelligent Computer Mathematics: 9th International Conference, AISC 2008, 15th Symposium, Calculemus 2008, 7th International Conference*. Birmingham, UK, July 2008, pp. 217–231.

[3] Jade Alglave, Alastair F. Donaldson, Daniel Kroening, and Michael Tautschnig. "Making Software Verification Tools Really Work". In: *Automated Technology for Verification and Analysis: 9th International Symposium*. Taipei, Taiwan: Springer Berlin Heidelberg, Oct. 2011, pp. 28–42.

[4] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. "An Empirical Evaluation of Portfolios Approaches for Solving CSPs". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 10th International Conference*. Yorktown Heights, NY, USA, May 2013, pp. 316–324.

[5] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. "Portfolio approaches for constraint optimization problems". In: *Annals of Mathematics and Artificial Intelligence* 76.1 (2016), pp. 229–246.

[6] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. "SUNNY-CP: A Sequential CP Portfolio Solver". In: *ACM Symposium on Applied Computing*. Salamanca, Spain, Apr. 2015, pp. 1861–1867.

[7] Michael Ameri and Carlo A. Furia. "Why Just Boogie?" In: *Integrated Formal Methods: 12th International Conference, IFM 2016*. Reykjavik, Iceland, June 2016, pp. 79–95.

[8] David Aspinall and Cezary Kaliszyk. "Towards Formal Proof Metrics". In: *Fundamental Approaches to Software Engineering: 19th International Conference, FASE 2016*. Eindhoven, The Netherlands, Apr. 2016, pp. 325–341.

[9] Saïd Assar, Markus Borg, and Dietmar Pfahl. "Using text clustering to predict defect resolution time: a conceptual replication and

an evaluation of prediction accuracy". In: *Empirical Software Engineering* 21.4 (2016), pp. 1437–1475.

[10] Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Jarvisalo, and Carsten Sinz. *Proceedings of SAT Challenge 2012; Solver and Benchmark Descriptions*. Tech. rep. 2. Helsinki, Finland: University of Helsinki, 2012.

[11] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. "The Spec# Programming System: An Overview". In: *Construction and Analysis of Safe, Secure and Interoperable Smart devices*. Marseille, France, Mar. 2004, pp. 49–69.

[12] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects: 4th International Symposium*. Amsterdam, The Netherlands, Nov. 2005, pp. 364–387.

[13] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`. 2016.

[14] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. 2010. URL: `http://www.smt-lib.org`.

[15] Clark Barrett and Cesare Tinelli. "CVC3". In: *Computer Aided Verification*. Berlin, Germany, July 2007, pp. 298–302.

[16] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *Computer Aided Verification*. Snowbird, UT, USA, July 2011, pp. 171–177.

[17] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer-Verlag Berlin Heidelberg, 2010.

[18] Dirk Beyer. "Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Eindhoven, The Netherlands, Apr. 2016, pp. 887–904.

[19] Dirk Beyer. "Status Report on Software Verification". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Grenoble, France, Apr. 2014, pp. 373–388.

[20] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Benchmarking and Resource Measurement". In: *Model Checking Software - 22nd International Symposium, SPIN 2015*. Stellenbosch, South Africa, Aug. 2015, pp. 160–178.

[21] Dirk Beyer, Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. "Evaluating Software Verification Systems: Benchmarks and Competitions (Dagstuhl Reports 14171)". In: *Dagstuhl Reports* 4.4 (2014). DOI: 10.4230/DagRep.4.4.1.

[22] Christopher M. Bishop. *Pattern recognition and machine learning*. New York, USA: Springer, 2006.

[23] Jasmin Christian Blanchette and Andrei Paskevich. "TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism". In: *CADE-24: 24th International Conference on Automated Deduction*. Lake Placid, NY, USA, June 2013, pp. 414–420.

[24] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. "A Learning-Based Fact Selector for Isabelle/HOL". in: *Journal of Automated Reasoning* 57.3 (2016), pp. 219–244.

[25] François Bobot and Andrei Paskevich. "Expressing Polymorphic Types in a Many-Sorted Language". In: *Frontiers of Combining Systems: 8th International Symposium, FroCoS 2011*. Saarbrücken, Germany, Oct. 2011, pp. 87–102.

[26] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. "Let's verify this with Why3". In: *International Journal on Software Tools for Technology Transfer* 17.6 (2015), pp. 709–727.

[27] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. "Preserving User Proofs across Specification Changes". In: *Verified Software: Theories, Tools, Experiments: 5th International Conference*. Menlo Park, CA, USA, May 2013, pp. 191–201.

[28] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 User Manual*. 0.87.2. 2016. URL: http://why3.lri.fr/doc-0.87.2/.

[29] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. "Why3: Shepherd your herd of provers". In: *Workshop on Intermediate Verification Languages*. Wroclaw, Poland, Aug. 2011, pp. 53–64.

[30] Andreas Bollin. "Is There Evolution Before Birth? Deterioration Effects of Formal Z Specifications". In: *Formal Methods and Software Engineering: 13th International Conference on Formal Engineering Methods*. Durham, UK, Oct. 2011, pp. 66–81.

[31] Hanen Borchani, Gherardo Varando, Concha Bielza, and Pedro Larranaga. "A survey on multi-output regression". In: *Data Mining And Knowledge Discovery* 5.5 (2015), pp. 216–233.

[32] Thorsten Bormer et al. "The COST IC0701 Verification Competition 2011". In: *Formal Verification of Object-Oriented Software*. Torino, Italy, Oct. 2011, pp. 3–21.

[33] Jean-Louis Boulanger. "Atelier B". in: *Formal Methods Applied to Complex Systems*. New York, USA: John Wiley & Sons, 2014, pp. 35–46.

[34] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. "VeriT: An Open, Trustable and Efficient SMT-Solver". In: *22nd International Conference on Automated Deduction*. Montreal, Canada, Aug. 2009, pp. 151–156.

[35] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32.

[36] Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. *Classification and Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series. Taylor & Francis, 1984.

[37] Alan Bundy, Dieter Hutter, Cliff B. Jones, and J Strother Moore. "AI meets Formal Software Development (Dagstuhl Seminar 12271)". In: *Dagstuhl Reports* 2.7 (2012), pp. 1–29. DOI: 10.4230/DagRep.2.7.1.

[38] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. "An overview of JML tools and applications". In: *International Journal on Software Tools for Technology Transfer* 7.3 (2005), pp. 212–232.

[39] Jordi Cabot and Ernest Teniente. "A metric for measuring the complexity of OCL expressions". In: *Workshop on Model Size Metrics (co-located with MODELS 2006)*. Genova, Italy, Oct. 2006.

[40] Robert N. Charette. "Why Software Fails [Software Failure]". In: *IEEE Spectr.* 42.9 (Sept. 2005), pp. 42–49.

[41] Shyam R. Chidamber and Chris F. Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493.

[42] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. "TIP: Tons of Inductive Problems". In: *Intelligent Computer Mathematics: International Conference*. Washington, DC, USA, July 2015, pp. 333–337.

[43] Tony Clark and Jos B. Warmer. *Object modeling with the OCL: the rationale behind the Object Constraint Language*. New York, USA: Springer, 2002.

[44] David R. Cok, Aaron Stump, and Tjark Weber. "The 2013 Evaluation of SMT-COMP and SMT-LIB". in: *Journal of Automated Reasoning* 55.1 (2015), pp. 61–90.

[45] Sylvain Conchon and Évan Contejean. *The Alt-Ergo automatic theorem prover*. 2008. URL: http://alt-ergo.lri.fr/.

[46] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". In: *Machine Learning* 20.3 (1995), pp. 273–297.

[47] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Budapest, Hungary, Mar. 2008, pp. 337–340.

[48] David Delahaye, Catherine Dubois, Claude Marché, and David Mentré. "The BWare Project: Building a Proof Platform for the Automated Verification of B Proof Obligations". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th International Conference, ABZ 2014*. Toulouse, France, June 2014, pp. 290–293.

[49] Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger. "Empirical Software Metrics for Benchmarking of Verification Tools". In: *Computer Aided Verification*. San Francisco, CA, USA, July 2015, pp. 561–579.

[50] Florent de Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. "Assisted Verification of Elementary Functions Using Gappa". In: *ACM Symposium on Applied Computing*. 2006, pp. 1318–1322.

[51] P. Domingos. *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. Basic Books, 2015.

[52] Hazel Duncan. "The Use of Data-Mining for the Automatic Formation of Tactics". PhD thesis. UK: School of Informatics, University of Edinburgh, 2007. URL: https://www.era.lib.ed.ac.uk/bitstream/handle/1842/1768/hazelthesis.pdf.

[53] Bruno Dutertre and Leonardo de Moura. *The Yices SMT Solver*. 2006. URL: http://yices.csl.sri.com/papers/tool-paper.pdf.

[54] Michael Färber and Cezary Kaliszyk. "Random Forests for Premise Selection". In: *Frontiers of Combining Systems: 10th International Symposium*. Wroclaw, Poland, Sept. 2015, pp. 325–340.

[55] Colin Farquhar, Gudmund Grov, Andrew Cropper, Stephen Muggleton, and Alan Bundy. "Typed meta-interpretive learning for proof strategies". In: *Late Breaking Papers of Inductive Logic Programming 2015*. Kyoto, Japan, Aug. 2015. URL: http://ceur-ws.org/Vol-1636/paper-02.pdf.

[56] Norman E. Fenton and Shari L. Pfleeger. *Software metrics: a rigorous and practical approach*. 2nd. Boston;London; PWS Pub, 1997.

[57] Jean-Christophe Filliâtre. “Deductive software verification”. In: *International Journal on Software Tools for Technology Transfer* 13.5 (2011), pp. 397–403.

[58] Jean-Christophe Filliâtre and Andrei Paskevich. “Why3 — Where Programs Meet Provers”. In: *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013.* Rome, Italy, Mar. 2013, pp. 125–128.

[59] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. “Malware Analysis and Classification: A Survey”. In: *Journal of Information Security* 5 (2014), pp. 56–64. DOI: 10.4236/jis.2014.52006.

[60] Thomas C. Hales et al. *A formal proof of the Kepler conjecture*. 2015. arXiv: 1501.02155.

[61] Felicia Halim. “Evaluate and Benchmark Arís”. MA thesis. National University of Ireland Maynooth, July 2014. URL: http://eprints.maynoothuniversity.ie/5341/.

[62] Andrew Healy, Rosemary Monahan, and James F. Power. “Evaluating the Use of a General-purpose Benchmark Suite for Domain-specific SMT-solving”. In: *31st Annual ACM Symposium on Applied Computing*. Pisa, Italy, May 2016, pp. 1558–1561.

[63] Andrew Healy, Rosemary Monahan, and James F. Power. “Predicting SMT Solver Performance for Software Verification”. In: Proceedings of the Third Workshop on *Formal Integrated Development Environment*. Limassol, Cyprus, 2017, pp. 20–37.

[64] Jónathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. “Proof-Pattern Recognition and Lemma Discovery in ACL2”. In: *Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference*. Stellenbosch, South Africa, Dec. 2013, pp. 389–406.

[65] Arthur E. Hoerl and Robert W. Kennard. “Ridge Regression: Biased Estimation for Nonorthogonal Problems”. In: *Technometrics* 42.1 (2000), pp. 80–86.

[66] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. *A practical guide to support vector classification*. 2003. URL: http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf.

[67] Chih-Wei Hsu and Chih-Jen Lin. “A comparison of methods for multiclass support vector machines”. In: *IEEE Transactions on Neural Networks* 13.2 (2002), pp. 415–425.

[68] Chenn-Jung Huang, Yu-Wu Wang, Chih-Tai Guan, Heng-Ming Chen, and Jui-Jiun Jian. "Applications of Machine Learning to Resource Management in Cloud Computing". In: *International Journal of Modeling and Optimization* 3.2 (2013), pp. 148–152.

[69] Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. "VerifyThis 2012". In: *International Journal on Software Tools for Technology Transfer* 17.6 (2015), pp. 647–657.

[70] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. "Proteus: A Hierarchical Portfolio of Solvers and Transformations". In: *Integration of AI and OR Techniques in Constraint Programming: 11th International Conference*. Cork, Ireland, May 2014, pp. 301–317.

[71] Alexei Iliasov, Paulius Stankaitis, David Adjepon-Yamoah, and Alexander Romanovsky. "Rodin Platform Why3 Plug-In". In: *ABZ 2016: Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference*. Linz, Austria, May 2016, pp. 275–281.

[72] Mateja Jamnik, Manfred Kerber, and Martin Pollet. "Automatic Learning of Proof Methods in Proof Planning". In: *Logic Journal of IGPL* 11.6 (2003), pp. 647–673.

[73] Kalervo Järvelin. "IR Research: Systems, Interaction, Evaluation and Theories". In: *SIGIR Forum* 45.2 (2012), pp. 17–31.

[74] Cezary Kaliszyk and Josef Urban. "Learning-Assisted Automated Reasoning with Flyspeck". In: *Journal of Automated Reasoning* 53.2 (2014), pp. 173–213.

[75] Cezary Kaliszyk and Josef Urban. "Learning-assisted theorem proving with millions of lemmas". In: *Journal of Symbolic Computation* 69 (July 2015), pp. 109 –128.

[76] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. "Robust Statistical Methods for Empirical Software Engineering". In: *Empirical Software Engineering* (2016), pp. 1–52. DOI: 10.1007/s10664-016-9437-5.

[77] Vladimir Klebanov et al. "The 1st Verified Software Competition: Experience Report". In: *FM 2011: 17th International Symposium on Formal Methods*. Limerick, Ireland, June 2011, pp. 154–168.

[78] Gerwin Klein. "Proof Engineering Considered Essential". In: *FM 2014: Formal Methods: 19th International Symposium*. Singapore, May 2014, pp. 16–21.

[79] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. "Comprehensive Formal Verification of an OS Microkernel". In: *ACM Transactions on Computer Systems* 32.1 (Feb. 2014), 2:1–2:70.

[80] Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. "Machine Learning in Proof General: Interfacing Interfaces". In: *10th International Workshop On User Interfaces for Theorem Provers*. Bremen, Germany, July 2012, pp. 15–41.

[81] Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. "Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014". In: *7th International Symposium on Leveraging Applications*. Corfu, Greece, Oct. 2016, p. 16.

[82] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference*. Dakar, Senegal, Apr. 2010, pp. 348–370.

[83] K. Rustan M. Leino and Michał Moskal. "VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0". In: *Tools and Experiments Workshop at VSTTE*. Edinburgh, UK, Aug. 2010.

[84] Wei Li and Sallie Henry. "Object-oriented metrics that predict maintainability". In: *The Journal of Systems and Software* 23.2 (1993), pp. 111–122.

[85] David J Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge, UK: Cambridge Univ. Press, 2000.

[86] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. "Parallel SAT Solver Selection and Scheduling". In: *Principles and Practice of Constraint Programming: 18th International Conference*. Québec City, QC, Canada, Oct. 2012, pp. 512–526.

[87] Claude Marché and Yannick Moy. *The Jessie plugin for Deductive Verification in Frama-C*. 2.35. 2015. URL: http://krakatoa.lri.fr/jessie.pdf.

[88] Daniel Matichuk, Toby Murray, June Andronick, Ross Jeffery, Gerwin Klein, and Mark Staples. "Empirical study towards a leading indicator for cost of formal software verification". In: *37th International Conference on Software Engineering*. Vol. 1. Florence, Italy, May 2015, pp. 722–732.

[89] Thomas J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320.

[90] Thomas J. McCabe and Charles W. Butler. “Design Complexity Measurement and Testing”. In: *Communications of the ACM* 32.12 (1989), pp. 1415–1425.

[91] David Mentré, Claude Marché, Jean-Christophe Filliâtre, and Masashi Asuka. “Discharging Proof Obligations from Atelier B using Multiple Automated Provers”. In: *ABZ'2012 - 3rd International Conference on Abstract State Machines, Alloy, B and Z*. Pisa, Italy, June 2012, pp. 238–251.

[92] Tom M. Mitchell. *Machine Learning*. New York, USA: McGraw-Hill, 1997.

[93] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. New York, USA: Springer, 2002.

[94] Lawrence C. Paulson and Jasmine Christian Blanchette. “Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers”. In: *8th International Workshop on the Implementation of Logics*. Yogyakarta, Indonesia, Oct. 2010.

[95] Fabian Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[96] James F. Power and Brian A. Malloy. “Metric-based analysis of context-free grammars”. In: *8th International Workshop on Program Comprehension*. Limerick, Ireland, June 2000, pp. 171–178.

[97] J. R. Quinlan. “Induction of decision trees”. In: *Machine Learning* 1.1 (1986), pp. 81–106.

[98] Luis Reynoso, Marcela Genero, and Mario Piattini. “Towards a metric suite for OCL Expressions expressed within UML/OCL models”. In: *Journal of Computer Science and Technology* 4.1 (2004), pp. 38–44.

[99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. New York, USA: Pearson Higher Education, 2004.

[100] Forrest Shull, Janice Singer, and Dag I. K. Sjoberg. *Guide to advanced empirical software engineering*. London, UK: Springer, 2010.

[101] Liyan Song, Leandro L. Minku, and Xin Yao. “The Potential Benefit of Relevance Vector Machine to Software Effort Estimation”. In: *10th International Conference on Predictive Models in Software Engineering*. PROMISE '14. Turin, Italy, Sept. 2014, pp. 52–61.

[102] Mark Staples, Rafal Kolanski, Gerwin Klein, Corey Lewis, June Andronick, Toby Murray, Ross Jeffery, and Len Bass. "Formal Specifications Better Than Function Points for Code Sizing". In: *2013 International Conference on Software Engineering*. San Francisco, CA, USA, May 2013, pp. 1257–1260.

[103] Geoff Sutcliffe. *The TPTP Problem Library*. Tech. rep. Dept. of Computer Science, University of Miami, 2016. URL: www.cs.miami.edu/~tptp/TPTP/TR/TPTPTR.shtml.

[104] Geoff Sutcliffe. "The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0". In: *Journal of Automated Reasoning* 43.4 (2009), pp. 337–362.

[105] Geoff Sutcliffe and Christian Suttner. "Evaluating general purpose automated theorem proving systems". In: *Artificial Intelligence* 131.1-2 (2001), pp. 39–54.

[106] Geoff Sutcliffe and Christian Suttner. "The TPTP Problem Library". In: *Journal Automated Reasoning* 21.2 (Oct. 1998), pp. 177–203.

[107] Asma Tafat and Claude Marché. *Binary Heaps Formally Verified in Why3*. Research Report RR-7780. INRIA, Oct. 2011. URL: https://hal.inria.fr/inria-00636083.

[108] Varun Tulsian, Aditya Kanade, Rahul Kumar, Akash Lal, and Aditya V. Nori. "MUX: algorithm selection for software model checkers". In: *11th Working Conference on Mining Software Repositories*. Hydrabad, India, May 2014, pp. 132–141.

[109] Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum, and David Frazier. "Incremental Benchmarks for Software Verification Tools and Techniques". In: *Verified Software: Theories, Tools, Experiments: Second International Conference, VSTTE 2008*. Toronto, Canada, Oct. 2008, pp. 84–98.

[110] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. "A Concurrent Portfolio Approach to SMT Solving". In: *Computer Aided Verification, 21st International Conference, CAV 2009*. Grenoble, France, June 2009, pp. 715–720.

[111] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. New York, USA: Springer, 2012.

[112] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. London, UK: Prentice Hall, 1996.

[113] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. "Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors". In: *Theory and Applications of Satisfiability Testing – SAT 2012*. Trento, Italy, June 2012, pp. 228–241.

[114] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "SATzilla: Portfolio-based Algorithm Selection for SAT". in: *Journal of Artificial Intelligence Research* 32.1 (2008), pp. 565–606.

[115] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "SATzilla2012: Improved Algorithm Selection Based on Cost-sensitive Classification Models". In: *Proceedings of SAT Challenge 2012; Solver and Benchmark Descriptions*. System Description. Helsinki, Finland, 2012.

[116] Du Zhang and Jeffrey J. Tsai, eds. *Machine learning applications in software engineering*. Vol. 16. Series on software Engineering and Knowledge Engineering. Hackensack, N.J, USA: World Scientific, 2005.