Predicting SMT Solver Performance for Software Verification

Andrew Healy, Rosemary Monahan, and James F. Power

Dept. of Computer Science,
Maynooth University, Maynooth, Ireland
{ahealy,rosemary,jpower}@cs.nuim.ie
http://www.cs.nuim.ie/research/pop/

Abstract. The Why3 IDE and verification system facilitates the use of a wide range of Satisfiability Modulo Theories (SMT) solvers through a driver-based architecture. We present Where4: a portfolio-based approach to discharge Why3 proof obligations. We use data analysis and machine learning techniques on static metrics derived from program source code. Our approach benefits software engineers by providing a single utility to delegate proof obligations to the solvers most likely to return a useful result. It does this in a time-efficient way using existing Why3 and solver installations — without requiring low-level knowledge about SMT solver operation from the user.

Keywords: SMT, Software Verification, Why3, Performance Prediction

1 Introduction

The formal verification of software generally requires a software engineer to use a system of tightly integrated components. Such systems typically consist of an IDE that can accommodate both the implementation of a program and the specification of its formal properties. These two aspects of the program are then typically translated into the logical constructs of an intermediate language, forming a series of goals which must be proved in order for the program to be fully verified. These goals (or "proof obligations") must be formatted for the system's general-purpose back-end solver. Examples of systems which follow this model are Spec# [3] and Dafny [28] which use the Boogie [2] intermediate language and the Z3 [20] SMT solver.

Why3 [11] was developed as an attempt to make use of the wide spectrum of interactive and automated theorem proving tools and overcome the limitations of systems which rely on a single SMT solver. It provides a driver-based, extensible architecture to perform the necessary translations into the input formats of two dozen provers. With a wide choice of theorem-proving tools now available to the software engineer, the question of choosing the most appropriate tool for the task at hand becomes important. It is this question that Where4 answers.

As motivation for our approach, Table 1 presents the results from running the Why3 tool over the example programs included in the Why3 distribution (version

0.87.1), using eight SMT solvers at the back-end. Each Why3 file contains a number of theories requiring proof, and these in turn are broken down into a number of goals for the SMT solver; for the data in Table 1 we had 128 example programs, generating 289 theories, in turn generating 1048 goals. In Table 1 each row presents the data for a single SMT solver, and the three main data columns give data totalled on a per-file, per-theory and per-goal basis. Each of these three columns is further broken down to show the number of programs/theories/goals that were successfully solved, their percentage of the total, and the average time taken in seconds for each solver to return such a result. Program verification by modularisation construct is particularly relevant to the use of Why3 on the command line as opposed to through the IDE.

Table 1 also has a row for an imaginary "theoretical" solver, $\mathcal{TS}$, which corresponds to choosing the best (fastest) solver for each individual program, theory or goal. This solver performs significantly better than any individual solver, and gives an indication of the maximum improvement that could be achieved *if it was possible to predict in advance which solver was the best for a given program, theory or goal.* In general, the method of choosing from a range of solvers on an individual goal basis is called *portfolio-solving.* This technique has been successfully implemented in the SAT solver domain by SATzilla [37] and for model-checkers [21][36]. Why3 presents a unique opportunity to use a common input language to develop a portfolio SMT solver specifically designed for software verification.

The main contributions of this paper are:

1. The design and implementation of our 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 formulae.
4. Statistics on the performance of eight SMT solvers using a dataset of 1048 Why3 goals.

Section 2 describes how the data was gathered and discusses issues around the accurate measurement of results and timings. A comparison of prediction models forms the basis of Section 3 where a number of evaluation metrics are introduced. The Where4 tool is compared to a range of SMT tools and strategies in Section 5. The remaining sections present a review of additional related work and a summary of our conclusions.

2 System Overview and Data Preparation

Due to the diverse range of input languages used by software verification systems, a standardised benchmark repository of verification programs does not yet exist [8]. For our study we chose the 128 example programs included in the

Table 1. Results of running 8 solvers on the example Why3 programs with a timeout value of 10 seconds. In total our dataset contained 128 files, which generated 289 theories, which in turn generated 1048 goals. Also included is a theoretical solver $\mathcal{TS}$, 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
$\mathcal{TS}$	**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

Why3 distribution (version 0.87.1) as our corpus for training and testing purposes. The programs in this repository are written in WhyML, a dialect of ML with added specification syntax and verified libraries. Many of the programs are solutions to problems posed at software verification competitions such as VerifyThis [13], VSTTE [26] and COST [15]. Other programs are implementations of benchmarks proposed by the VACID-0 [29] initiative. It is our assumption that these programs are a representative software verification workload. Alternatives to this dataset are discussed in Section 6.

We used six current, general-purpose SMT solvers supported by Why3: Alt-Ergo [19] versions 0.95.2 and 4.4.1, CVC3 [6] ver. 2.4.1, CVC4 [4] ver. 1.4, veriT [16], ver. 201506[1], Yices [22] ver. 1.0.38[2], and Z3 [20] ver. 4.3.2 and 4.4.1. We expanded the range of solvers to eight by recording the results for two of the most recent major versions of two popular solvers - Alt-Ergo and Z3.

When a solver is sent a goal by Why3 it returns one of the five possible answers *Valid*, *Invalid*, *Unknown*, *Timeout* or *Failure*. As can be seen from Table 1 and Fig. 1, not all goals can be proved Valid or Invalid. Such goals usually require the use of an interactive theorem prover to discharge goals that require reasoning by induction. Sometimes a splitting transformation needs to be applied to simplify the goals before they are sent to the solver. Our tool 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 Section 2.1.

[1] The most recent version - 201506 - is not officially supported by *Why3* but is the only version available

[2] We did not use Yices2 as its lack of support for quantifiers makes it unsuitable for software verification

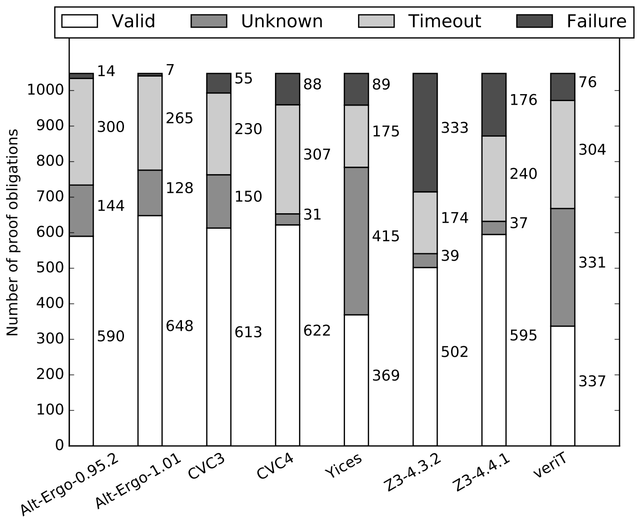

Fig. 1. The relative amount of Valid/Unknown/Timeout/Failure answers from the eight SMT solvers (with a timeout of 60 seconds). Note that no tool returned an answer of Invalid for any of the 1048 proof obligations.

2.1 Problem Quantification: predictor and response variables

Two sets of data need to be gathered in supervised machine learning [31]: the independent/predictor variables which are used as input for both training and testing phases, and the dependent/response variables which correspond to ground truths during training. Of the 128 programs in our dataset, 25% were held back for system evaluation (Section 5). The remaining 75% (corresponding to 96 WhyML programs, 768 goals) were used for training and 4-Fold cross-validation.

Independent/Predictor Variables Fig. 2 lists the predictor variables that were used in our study. All of these are (integer-valued) metrics that can be calculated by analysing a Why3 proof obligation, and are similar to the *Syntax* metadata category for proof obligations written in the TPTP format [34]. To construct a feature vector from each task sent to the solvers, we traverse the abstract syntax tree (AST) for each goal and lemma, counting the number of each syntactic feature we find on the way. We focus on goals and lemma as they produce proof obligations, with axioms and predicates providing a logical context.

Our feature extraction algorithm has similarities in this respect to the method used by Why3 for computing goal "shapes" [12]. These shape strings are used internally by Why3 as an identifying fingerprint. Across proof sessions, their use can limit the amount of goals in a file which need to be re-proved.

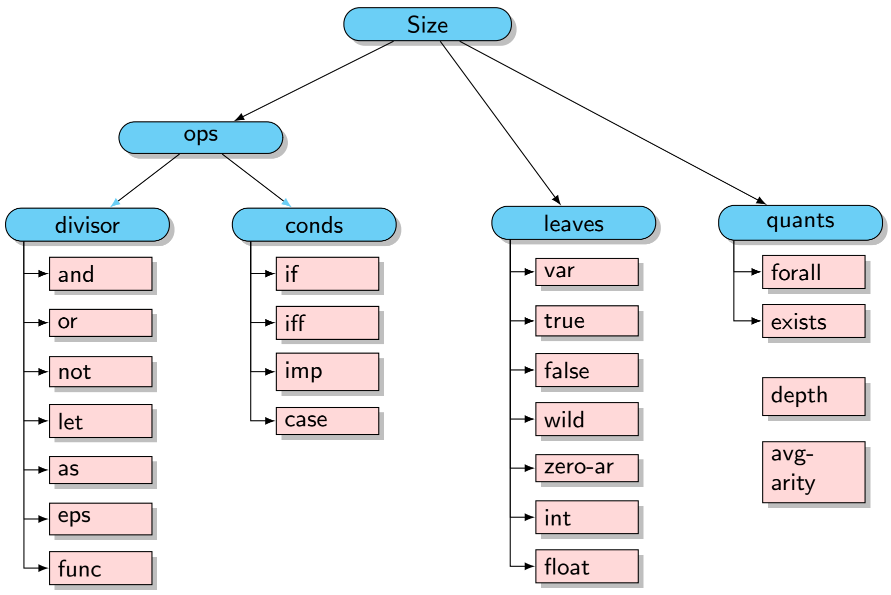

Fig. 2. Tree illustrating the Why syntactic features counted individually (*pink nodes*) while traversing the AST. The rectangles represent individual measures, and the rounded blue nodes represent metrics that are the sum of their children in the tree.

Dependent/Response Variables Our evaluation of the performance of a solver depends on two factors: the time taken to calculate that result, and whether or not the solver had actually proven the goal.

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 [9] framework was developed by the organisers of the SVCOMP [7] 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. To account for random errors in measurement introduced at each execution, we used the methodology described by Lilja [30] to obtain an approximation of the true mean time. A 90% confidence interval was used with an allowed error of ±3.5%.

By inspecting our data, we saw that most *Valid* and *Invalid* answers returned very quickly, with *Unknown* answers taking slightly longer, and *Failure/Timeout* responses taking longest. We took the relative utility of responses to be $\{Valid, Invalid\} > Unknown > \{Timeout, Failure\}$ which can be read as “it is better for a solver to return a *Valid* response than *Timeout*”, etc. A simple function allocates a cost to each solver S's response to each goal G:

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

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 backend 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 ascending cost.

Since our cost model depends on the time limit value chosen, we need to choose a value that does not favour any one solver. To establish a realistic time limit value, we find each solver's "Peter Principle Point" [35]. 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.

Fig. 3 shows the number of *Valid/Invalid/Unknown* results for each prover when given a time limit of 60 seconds. This value was chosen as an upper limit, since a time limit value of 60 seconds is not realistic for most software verification scenarios. Why3, for example, has a default time limit value of 5 seconds. From Fig. 3 we can see that the vast majority of useful responses are returned very quickly.

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. This threshold ranges from 7.35 secs (veriT) to 9.69 secs (Z3-4.3.2). Thus we chose a value of 10 seconds as a representative, realistic time limit that gives each solver a fair opportunity to return decent results.

3 Choosing a prediction model

Given a Why3 goal, a ranking of solvers can be obtained by sorting the cost for each solver. For unseen instances, two approaches to prediction can be used: (1) classification — predicting the final ranking directly — and (2) regression — predicting each solver's score individually and deriving a ranking from these predictions. With eight solvers, there are 8! possible rankings. Many of these rankings were observed very rarely or did not appear at all in the training data. Such an unbalanced dataset is not appropriate for accurate classification, leading us to pursue the regression approach.

Seven regression models were evaluated[3]: Linear Regression, Ridge Regression, K-Nearest Neighbours, Decision Trees, Random Forests (with and without discretisation) and the regression variant of Support Vector Machines. Table 2 shows the results for some of the best-performing models. Most models were evaluated with and without a weighting function applied to the training samples. Weighting is standard practice in supervised machine learning: each sample's weight was defined as the standard deviation of solver costs. This function was

[3] We used the Python Sci-kit Learn [33] implementations of these models

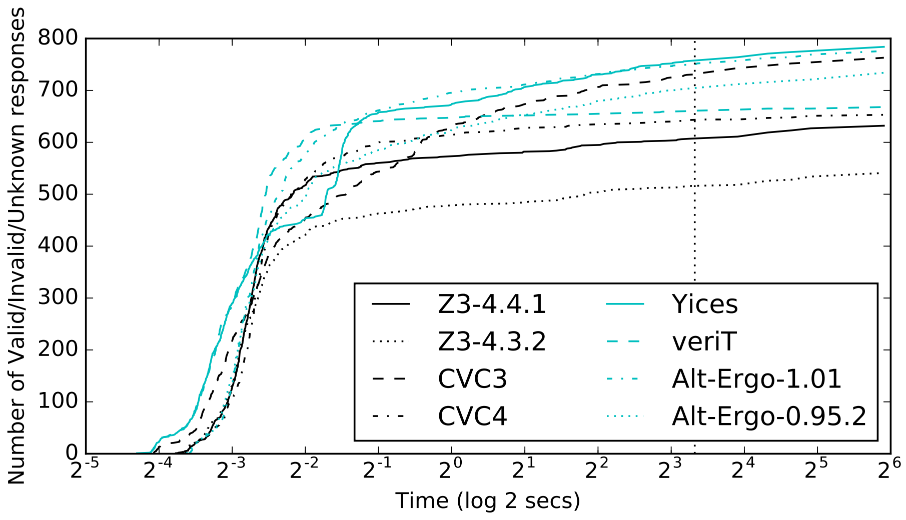

Fig. 3. The cumulative number of *Valid/Invalid/Unknown* responses for each solver. The plot uses a logarithmic scale on the time axis for increased clarity at the low end of the scale. The chosen timeout limit of 10 secs (*dotted vertical line*) includes 99% of each solver's useful responses

designed to give more importance to instances where there was a large difference in performance among the solvers.

Table 2 also shows three theoretical strategies in order to provide bounds for the prediction models. *Best* always chooses the best ranking of solvers and *Worst* always chooses the worst ranking (which is the reverse ordering to *Best*). *Random* is the average result of choosing every permutation of the eight solvers for each instance in the training set. We use this strategy to represent the user selecting SMT solvers at random without any consideration for goal characterisation or solver capabilities. A comparison to a *fixed* ordering of solvers for each goal is not made as any such ordering would be arbitrarily determined.

We note that the *Best* theoretical strategy of Table 2 is not directly comparable with the theoretical solver $\mathcal{TS}$ from Table 1. The two tables' average time columns are measuring different results: in contrast to $\mathcal{TS}$, *Best* will call each solver in turn, as will all the other models in Table 2, until a *Valid/Invalid* result is recorded (which it may never be). Thus Table 2's *Time* column shows the average *cumulative* time of each such sequence of calls, rather than the average time taken by the single *best* solver called by $\mathcal{TS}$.

3.1 Evaluating the prediction models

Table 2's *Time* column provides an overall estimate of the effectiveness of each prediction model. We can see that the discretised Random Forest method provides the best overall results for the solvers, yielding an average time of 14.92 seconds.

The second numeric column of Table 2 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 [24]. Here, emphasis is placed on correctly predicting items higher in the ranking. For a general ranking of length p, it is formulated as:

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

Here 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 0 impossible), the values in Table 2 are further normalised to the range [0..1] using the lower $nDCG$ bound for ranks of length 8 — found empirically to be 0.4394.

The third numeric column of Table 2 shows the R^2 score (or coefficient of determination), which is an established metric for evaluating 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. Table 2's fourth numeric column shows the MAE (Mean Average Error) — a ranking metric 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. Finally, the fifth numeric column of Table 2 shows the mean regression error (*Reg. error*) which measures the mean absolute difference in predicted solver costs to actual values.

Table 2. Comparing the seven prediction models and three theoretical strategies

	Time (secs)	**nDCG**	R^2	**MAE**	**Reg. error**
Best	12.63	1.00	-	0.00	0.00
Random	19.06	0.36	-	2.63	50.77
Worst	30.26	0.00	-	4.00	94.65
Random Forest	15.02	0.48	**0.28**	2.08	**38.91**
Random Forest (discretised)	**14.92**	0.48	-0.18	2.13	39.19
Decision Tree	15.80	0.50	0.11	2.06	43.12
K-Nearest Neighbours	15.93	**0.53**	0.16	**2.00**	43.41
Support Vector Regressor	15.57	0.47	0.14	2.26	47.45
Linear Regression	15.17	0.42	-0.16	2.45	49.25
Ridge	15.11	0.42	-0.15	2.45	49.09

3.2 Discussion: choosing a prediction model

An interesting feature of all the best-performing models in Table 2 is their ability to predict *multi-output* variables [14]. In contrast to the Support Vector model, for example, which must predict the cost for each solver individually, a multi-output model predicts 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.

After inspecting the results for all learning algorithms (summarised in Table 2), we can see that random forests [17] perform well, relative to other methods. They score highest for three out of 5 metrics (shown in bold) and have generally good scores in the others. Random forests are an ensemble extension of decision trees: random subsets of the training data are used to train each tree. For regression tasks, the set of predictions for each tree is averaged to obtain the forest's prediction. This method is designed to prevent over-fitting.

Based on the data in Table 2 we selected random forests as the choice of predictor to use in Where4.

4 Implementing Where4 in OCaml

Where4's interaction with Why3 is inspired by the use of machine learning in the Sledgehammer tool [10] which allows the use of SMT solvers in the interactive theorem prover Isabelle/HOL. We aspired to Sledgehammer's 'zero click, zero maintenance, zero overhead' philosophy in this regard: it should not interfere with a Why3 user's normal work-flow nor should it penalise those who do not use it.

We implement a "pre-solving" heuristic commonly used by portfolio solvers [1][37]: a single solver is called with a short time limit before feature extraction and solver rank prediction takes place. By using a good "pre-solver" at this initial stage, easily-proved instances are filtered with a minimum time overhead. We used a ranking of solvers based on the number of goals each could prove, using the data from Table 1. The highest-ranking solver installed locally is chosen as a "pre-solver". For the purposes of this paper, which assumes all 8 solvers are installed, this solver corresponds to Alt-Ergo version 1.01.

The random forest is fitted on the entire training set and encoded as a JSON file for legibility and modularity. This approach allows new trees and forests devised by the user (possibly using new SMT solvers or data) to replace our model. When the user installs Where4 locally, this JSON file is read and printed as an OCaml array. For efficiency, other important configuration information is compiled into OCaml data structures at this stage: e.g. the user's `why3.conf` file is read to determine the supported SMT solvers. All files are compiled and a native binary is produced. This only needs to be done once (unless the locally installed provers have changed).

The Where4 command-line tool has the following functionality:

1. Read in the WhyML/Why file and extract feature vectors from its goals.
2. Find the predicted costs for each of the 8 provers by traversing the random forest, using each goal's feature vector.
3. Sort the costs to produce a ranking of the SMT solvers.
4. Use the Why3 API to call each solver (if it is installed) in rank order.
5. Repeat 4 as necessary until a *Valid/Invalid* response is recorded or all installed solvers have been called.

If the user has selected that Where4 be available for use through Why3, the file which lets Why3 know about supported provers installed locally is modified to contain a new entry for the Where4 binary. A simple driver file (which just tells Why3 to use the Why logical language for encoding) is added to the drivers' directory. At this point, Where4 can be detected by Why3, and then used at the command line, through the IDE or by the OCaml API just like any other supported solver.

5 Evaluating **Where4**'s performance on test data

The evaluation of Where4 was carried out on a test set of 32 WhyML files, 77 theories, 263 goals (representing 25% of the entire dataset). This section is guided by the following three evaluation criteria:

Table 3. Number of files, theories and goals proved by each strategy and individual solver. The percentage this represents of the total 32 files, 77 theories and 263 goals and the average time are also shown.

	File			**Theory**			**Goal**		
	# proved	% proved	Avg time	# proved	% proved	Avg time	# proved	% proved	Avg time
Where4	11	34.4%	1.39	44	57.1%	0.99	203	77.2%	1.98
Best			0.25			0.28			0.37
Random			4.19			4.02			5.70
Worst			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

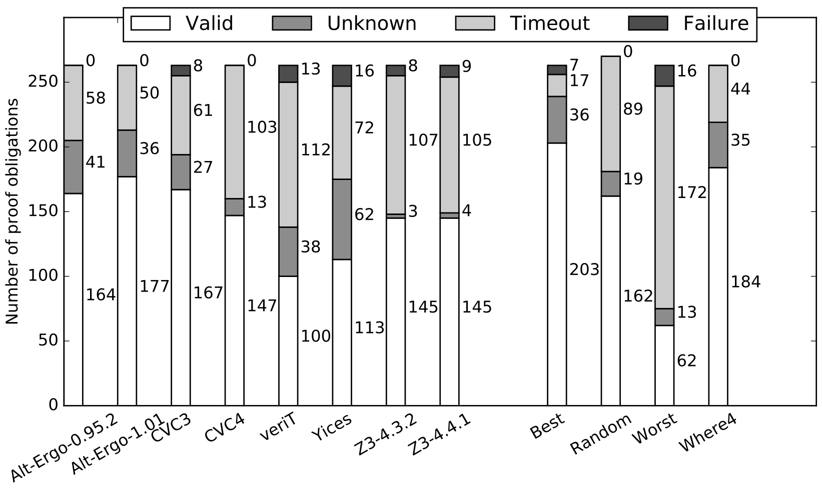

Fig. 4. The relative amount of Valid/Unknown/Timeout/Failure answers from the eight SMT solvers. Shown on the right are results obtainable by using the top solver (only) with the 3 ranking strategies and the Where4 predicted ranking (with Alt-Ergo-1.01 pre-proving).

How does Where4 perform in comparison to the 8 SMT solvers under consideration? When each solver in Where4's ranking sequence is run on each goal, the maximum amount of files, theories and goals are provable. As Table 3 shows, the difference between Where4 and our set of reference theoretical strategies (*Best, Random*, and *Worst*) is the amount of time taken to return the *Valid/Invalid* result. Compared to the 8 SMT provers, 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.

Unfortunately, the average time taken to solve each of these goals is high when compared to the 8 SMT provers. 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 and time-out for all 8 solvers in sequence, whereas each individual solver does this just once. Thus, while having access to more solvers allows more goals to be proved, there is also a time penalty to portfolio-based solvers in these circumstances.

At the other extreme, we could limit the portfolio solver to just using the best predicted individual solver, eliminating the multiple time-out overhead. Fig. 4 shows that the effect of this is to reduce the number of goals provable by Where4, though this is still more than the best-performing individual SMT solver, Alt-Ergo-1.01.

To calibrate this cost of Where4 against the individual SMT solvers, we introduce the notion of a *cost threshold*: using this strategy, after pre-proving,

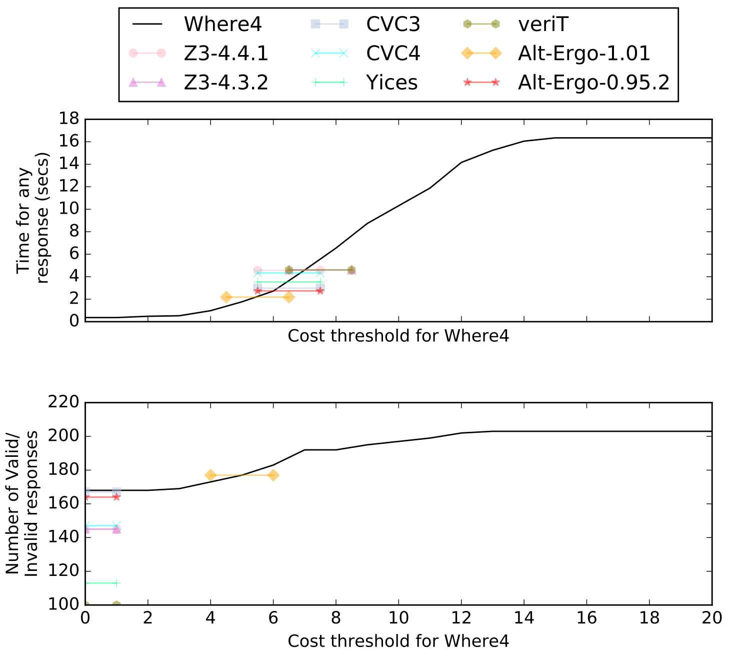

Fig. 5. The effect of using a cost threshold. (*top*) The average time taken for Where4 to return an answer compared to 8 SMT solvers. (*bottom*) The number of Valid/Invalid answers returned by Where4 compared to 8 SMT solvers. For the 7 solvers other than Alt-Ergo-1.01, the number of provable goals is indicated by a mark on the y-axis rather than an intersection with Where4's results.

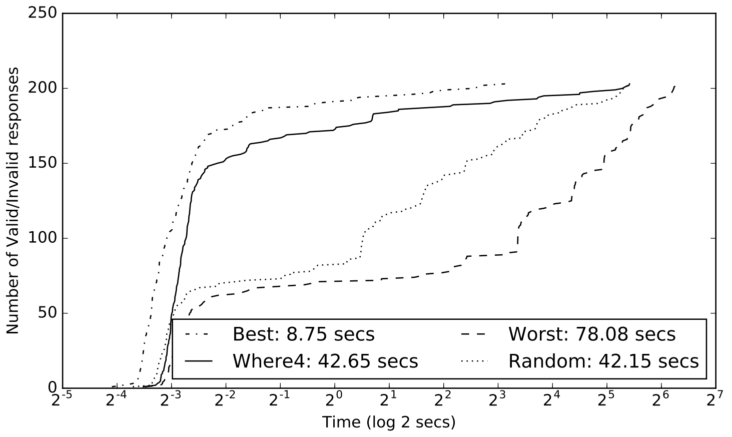

Fig. 6. The cumulative time each theoretical strategy, and Where4 to return all *Valid/Invalid* answers in the test dataset of 263 goals

solvers with a predicted cost above this threshold are not called. If no solver's cost is predicted below the threshold, the pre-prover's result is returned.

Fig. 5 shows the effect of varying this threshold, expressed in terms of the average execution time (top graph) and the number of goals solved (bottom graph). As we can see from both graphs in Fig. 5, for the goals in the test set a threshold of 7 for the cost function allows Where4 to prove more goals than any single solver, in a time approximately equal to the four slower solvers (CVC4, veriT and both versions of Z3).

How does Where4 perform in comparison to the 3 theoretical ranking strategies? Fig. 6 compares the cumulative time taken for Where4 and the 3 ranking strategies to return the 203 valid answers in the test set. Although both Where4 and *Random* finish at approximately the same time, Where4 is significantly faster for returning *Valid/Invalid* answers. Where4's solid line is more closely correlated to *Best*'s rate of success than the erratic rate of the *Random* strategy. *Best*'s time result shows the capability of a perfect-scoring learning strategy. It is motivation to further improve Where4 in the future.

What is the time overhead of using Where4 to prove Why3 goals? The timings for Where4 in all plots and tables are based solely on the performance of the constituent solvers (the measurement of which is discussed in Sec. 2.1). 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 file. On a per goal basis, this is equivalent to an increase in 0.056 seconds.

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. Future work will look at bypassing this step for WhyML files while still allowing files to be proved on an individual theory and goal basis.

5.1 Threats to Validity

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. 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. There were far more *Valid* responses than any other response. No goal in our dataset returned an answer of *Invalid* on any of the 8 solvers. This is a serious problem as Where4 would not be able to recognize such a goal in real-world use. In future work we hope to use the TPTP benchmark library to remedy these issues. The benchmarks in this library come from a diverse range of contributors working in numerous problem domains [35] and are not as specific to software verification as the Why3 suite of examples.

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. 2.1). 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 [21][36]). 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 [2] intermediate language.

External The generalisability of our results is limited by the fact that all dependent variables were measured on a single machine.[4] 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), 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 Comparison with Related Work

Comparing verification systems: The need for a standard set of benchmarks for the diverse range of software systems is a recurring issue in the literature [8]. The benefits of such a benchmark suite are identified by the SMTLIB [5] 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 [18][7]. The TPTP (Thousands of

[4] All data collection was conducted on a 64-bit machine running Ubuntu 14.04 with a dual-core Intel i5-4250U CPU and 16GB of RAM.

Problems for Theorem Provers) project [34] has similar aims but a wider scope: targeting 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 [35].

Portfolio solvers: Portfolio-solving approaches have been implemented successfully in the SAT domain by SATzilla [37] and the constraint satisfaction / optimisation community by tools such as CPHydra [32] and sunny-cp [1]. Numerous studies have used the SVCOMP [7] benchmark suite of C programs for model checkers to train portfolio solvers [36][21]. These particular studies have been predicated on the use of Support Vector Machines (SVM) with only a cursory use of linear regression [36]. In this respect, our project represents a more wide-ranging treatment of the various prediction models available for portfolio solving. The need for a strategy to delegate Why3 goals to appropriate SMT solvers is stated in recent work looking at verification systems on cloud infrastructures [23].

Machine Learning in Formal Methods: The FlySpec [25] corpus of proofs has been the basis for a growing number of tools integrating interactive theorem provers with machine-learning based fact-selection. The MaSh engine in Sledgehammer [10] is a related example. It uses a Naive Bayes algorithm and clustering to select facts based on syntactic similarity. Unlike Where4, MaSh uses a number of metrics to measure the *shape* of goal formulæas features. The weighting of features uses an inverse document frequency (IDF) algorithm. ML4PG (Machine Learning for Proof General) [27] also uses clustering techniques to guide the user for interactive theorem proving.

Our work adds to the literature by applying a portfolio-solving approach to SMT solvers. We conduct a wider comparison of learning algorithms than other studies which mostly use either SVMs or clustering. Unlike the interactive theorem proving tools mentioned above, Where4 is specifically suited to software verification through its integration with the Why3 system.

7 Conclusion and Future Work

We have presented a strategy to choose appropriate SMT solvers based on Why3 syntactic features. Users without any knowledge of SMT solvers can prove a greater number of goals in a shorter amount of time by delegating to Where4 than by choosing solvers at random. Although some of Where4's results are disappointing, we believe that the Why3 platform has great potential for machine-learning based portfolio-solving. We are encouraged by the performance of a theoretical *Best* strategy and the convenience that such a tool would give Why3 users.

The number of potential directions for this work is large: parallel solving, the use of an initial solver to filter trivial instances before feature extraction, larger and more generic datasets, etc. The TPTP repository represents a large source of proof obligations which can be translated into the Why logic language. The number of goals provable by Where4 could be increased by identifying which goals need to be simplified in order to be tractable for an SMT solver. Splitting transforms would also increase the number of goals for training data: from 1048 to 7489 through the use of the `split_goal_wp` transform, for example. An interesting direction for this work could be the identification of the appropriate transformations. Also, we will continue to improve the efficiency of Where4 when used as a Why3 solver and investigate the use of a minimal benchmark suite which can be used to train the model using new SMT solvers and theorem provers installed locally.

Data related to this paper is hosted at `github.com/ahealy19/F-IDE-2016`. Where4 is hosted at `github.com/ahealy19/Where4`.

Acknowledgments. This project is being carried out with funding provided by Science Foundation Ireland under grant number 11/RFP.1/CMS/3068

References

1. Roberto Amadini, Maurizio Gabbrielli & Jacopo Mauro (2015): *SUNNY-CP: A Sequential CP Portfolio Solver.* In: *ACM Symposium on Applied Computing*, Salamanca, Spain, pp. 1861–1867, doi:10.1145/2695664.2695741.
2. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs & K. Rustan M. Leino (2005): *Boogie: A Modular Reusable Verifier for Object-Oriented Programs.* In: *Formal Methods for Components and Objects: 4th International Symposium*, Amsterdam, The Netherlands, pp. 364–387, doi:10.1007/11804192_17.
3. Mike Barnett, K. Rustan M. Leino & Wolfram Schulte (2004): *The Spec# Programming System: An Overview.* In: *Construction and Analysis of Safe, Secure and Interoperable Smart devices*, Marseille, France, pp. 49–69, doi:10.1007/978-3-540-30569-9_3.
4. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds & Cesare Tinelli (2011): CVC4. In: *Computer Aided Verification*, Snowbird, UT, USA, pp. 171–177.
5. Clark Barrett, Aaron Stump & Cesare Tinelli (2010): *The Satisfiability Modulo Theories Library (SMT-LIB).* Available at `http://www.smt-lib.org`.
6. Clark Barrett & Cesare Tinelli (2007): *CVC3.* In: *Computer Aided Verification*, Berlin, Germany, pp. 298–302.
7. Dirk Beyer (2014): *Status Report on Software Verification.* In: *Tools and Algorithms for the Construction and Analysis of Systems*, Grenoble, France, pp. 373–388, doi:10.1007/978-3-642-54862-8_25.
8. Dirk Beyer, Marieke Huisman, Vladimir Klebanov & Rosemary Monahan (2014): *Evaluating Software Verification Systems: Benchmarks and Competitions (Dagstuhl Reports 14171).* *Dagstuhl Reports* 4(4), doi:10.4230/DagRep.4.4.1.
9. Dirk Beyer, Stefan Löwe & Philipp Wendler (2015): *Benchmarking and Resource Measurement.* In: *Model Checking Software - 22nd International Symposium, SPIN 2015*, Stellenbosch, South Africa, pp. 160–178, doi:10.1007/978-3-319-23404-5_12.
10. Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein & Josef Urban (2016): *A Learning-Based Fact Selector for Isabelle/HOL.* *Journal of Automated Reasoning*, pp. 1–26, doi:10.1007/s10817-016-9362-8.

11. François Bobot, Jean-Christophe Filliâtre, Claude Marché & Andrei Paskevich (2011): *Why3: Shepherd your herd of provers.* In: *Workshop on Intermediate Verification Languages*, Wroclaw, Poland, pp. 53–64.
12. François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond & Andrei Paskevich (2013): *Preserving User Proofs across Specification Changes.* In: *Verified Software: Theories, Tools, Experiments: 5th International Conference*, Menlo Park, CA, USA, pp. 191–201, doi:`10.1007/978-3-642-54108-7_10`.
13. François Bobot, Jean-Christophe Filliâtre, Claude Marché & Andrei Paskevich (2015): *Let's verify this with Why3. International Journal on Software Tools for Technology Transfer* 17(6), pp. 709–727, doi:`10.1007/s10009-014-0314-5`.
14. H. Borchani, G. Varando, C. Bielza & P. Larranaga (2015): *A survey on multi-output regression. Data Mining And Knowledge Discovery* 5(5), pp. 216–233.
15. Thorsten Bormer, Marc Brockschmidt, Dino Distefano, Gidon Ernst, Jean-Christophe Filliâtre, Radu Grigore, Marieke Huisman, Vladimir Klebanov, Claude Marché, Rosemary Monahan, Wojciech Mostowski, Nadia Polikarpova, Christoph Scheben, Gerhard Schellhorn, Bogdan Tofan, Julian Tschannen & Mattias Ulbrich (2011): *The COST IC0701 Verification Competition 2011.* In: *Formal Verification of Object-Oriented Software*, Torino, Italy, pp. 3–21.
16. Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe & Pascal Fontaine (2009): *veriT: An Open, Trustable and Efficient SMT-Solver.* In: *22nd International Conference on Automated Deduction*, Montreal, Canada, pp. 151–156, doi:`10.1007/978-3-642-02959-2_12`.
17. Leo Breiman (2001): *Random Forests. Machine Learning* 45(1), pp. 5–32, doi:`10.1023/A:1010933404324`.
18. David R. Cok, Aaron Stump & Tjark Weber (2015): *The 2013 Evaluation of SMT-COMP and SMT-LIB. Journal of Automated Reasoning* 55(1), pp. 61–90, doi:`10.1007/s10817-015-9328-2`.
19. Sylvain Conchon & Évan Contejean (2008): *The Alt-Ergo automatic theorem prover.* Available at `http://alt-ergo.lri.fr/`.
20. Leonardo De Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver.* In: *Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, Hungary, pp. 337–340.
21. Yulia Demyanova, Thomas Pani, Helmut Veith & Florian Zuleger (2015): *Empirical Software Metrics for Benchmarking of Verification Tools.* In: *Computer Aided Verification*, San Francisco, CA, USA, pp. 561–579, doi:`10.1007/978-3-319-21690-4_39`.
22. Bruno Dutertre & Leonardo de Moura (2006): *The Yices SMT Solver.* Available at `http://yices.csl.sri.com/papers/tool-paper.pdf`.
23. Alexei Iliasov, Paulius Stankaitis, David Adjepon-Yamoah & Alexander Romanovsky (2016): *Rodin Platform Why3 Plug-In.* In: *ABZ 2016: Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference*, Linz, Austria, pp. 275–281, doi:`10.1007/978-3-319-33600-8_21`.
24. Kalervo Järvelin (2012): *IR Research: Systems, Interaction, Evaluation and Theories. SIGIR Forum* 45(2), pp. 17–31, doi:`10.1145/2093346.2093348`.
25. Cezary Kaliszyk & Josef Urban (2014): *Learning-Assisted Automated Reasoning with Flyspeck. Journal of Automated Reasoning* 53(2), pp. 173–213, doi:`10.1007/s10817-014-9303-3`.
26. Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan

Tobies, Thomas Tuerk, Mattias Ulbrich & Benjamin Weiß (2011): *The 1st Verified Software Competition: Experience Report.* In: *FM 2011: 17th International Symposium on Formal Methods*, Limerick, Ireland, pp. 154–168, doi:10.1007/978-3-642-21437-0_14.

27. Ekaterina Komendantskaya, Jónathan Heras & Gudmund Grov (2012): *Machine Learning in Proof General: Interfacing Interfaces.* In: *10th International Workshop On User Interfaces for Theorem Provers*, Bremen, Germany, pp. 15–41, doi:10.4204/EPTCS.118.2.
28. K. Rustan M. Leino (2010): *Dafny: An Automatic Program Verifier for Functional Correctness.* In: *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference*, Dakar, Senegal, pp. 348–370, doi:10.1007/978-3-642-17511-4_20.
29. K. Rustan M. Leino & Michał Moskal (2010): *VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0.* In: *Tools and Experiments Workshop at VSTTE*.
30. David J Lilja (2000): *Measuring computer performance: a practitioner's guide.* Cambridge Univ. Press, Cambridge, UK.
31. Tom M. Mitchell (1997): *Machine Learning.* McGraw-Hill, New York, USA.
32. Eoin OMahony, Emmanuel Hebrard, Alan Holland, Conor Nugent & Barry OSullivan (2008): *Using case-based reasoning in an algorithm portfolio for constraint solving.* In: *Irish Conference on Artificial Intelligence and Cognitive Science*, pp. 210–216.
33. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot & E. Duchesnay (2011): *Scikit-learn: Machine Learning in Python.* *Journal of Machine Learning Research* 12, pp. 2825–2830.
34. Geoff Sutcliffe & Christian Suttner (1998): *The TPTP Problem Library.* *Journal Automated Reasoning* 21(2), pp. 177–203, doi:10.1023/A:1005806324129.
35. Geoff Sutcliffe & Christian Suttner (2001): *Evaluating general purpose automated theorem proving systems.* *Artificial Intelligence* 131(1-2), pp. 39–54, doi:10.1016/S0004-3702(01)00113-8.
36. Varun Tulsian, Aditya Kanade, Rahul Kumar, Akash Lal & Aditya V. Nori (2014): *MUX: algorithm selection for software model checkers.* In: *11th Working Conference on Mining Software Repositories*, Hydrabad, India, pp. 132–141.
37. Lin Xu, Frank Hutter, Holger H. Hoos & Kevin Leyton-Brown (2008): *SATzilla: Portfolio-based Algorithm Selection for SAT.* *Journal of Artificial Intelligence Research* 32(1), pp. 565–606.