MUX: Algorithm Selection for Software Model Checkers

Varun Tulsian* Aditya Kanade
Indian Institute of Science
(* Now at Walmart Labs)
{varuntulsian,kanade}@csa.iisc.ernet.in

Rahul Kumar Akash Lal Aditya V. Nori
Microsoft Research India
{rahulku,akashl,adityan}@microsoft.com

ABSTRACT

With the growing complexity of modern day software, software model checking has become a critical technology for ensuring correctness of software. As is true with any promising technology, there are a number of tools for software model checking. However, their respective performance trade-offs are difficult to characterize accurately – making it difficult for practitioners to select a suitable tool for the task at hand. This paper proposes a technique called MUX that addresses the problem of selecting the most suitable software model checker for a given input instance. MUX performs machine learning on a repository of software verification instances. The algorithm selector, synthesized through machine learning, uses structural features from an input instance, comprising a program-property pair, at runtime and determines which tool to use.

We have implemented MUX for Windows device drivers and evaluated it on a number of drivers and model checkers. Our results are promising in that the algorithm selector not only avoids a significant number of timeouts but also improves the total runtime by a large margin, compared to any individual model checker. It also outperforms a portfolio-based algorithm selector being used in Microsoft at present. Besides, MUX identifies structural features of programs that are key factors in determining performance of model checkers.

Categories and Subject Descriptors:
D.2.4 [*Software Engineering*]: Software/Program Verification–Model checking; D.2.8 [*Software Engineering*]: Metrics–Performance measures; I.2.6 [*Artificial Intelligence*]: Learning–Parameter learning

General Terms:
Performance, Reliability, Verification

Keywords:
Algorithm selection, machine learning, software model checking

1. INTRODUCTION

With the growing complexity of modern day software, it is becoming increasingly challenging to ensure correctness of software through manual and informal processes. *Software model checking* is a technique for proving properties of software or detecting buggy execution traces that violate the properties. Thus, they can not only verify correctness of software but can also generate debugging information, and have been hugely successful in practice (*e.g.*, [45, 6]). Even though a large research community continues to work tirelessly on improving performance of such tools, very little attention has been given to the difficulties faced by *practitioners* who focus on deploying them in the real-world.

To increase the adoption of these techniques in practice, there are two major aspects that require attention: (1) design of specification languages that are easy to use and (2) support for selecting the right tool for a task at hand. The need for the former is served to an extent with the design of both graphical (*e.g.*, UML [20]) and textual (*e.g.*, SDL [26], JML [34], SLIC [8]) specification languages. As is true with any promising technology, there are a number of competing tools for software model checking (*e.g.*, [45, 19, 36, 7, 38, 32, 24]). They use a variety of algorithmic techniques and offer different trade-offs in terms of efficiency and completeness. For instance, one tool may succeed in model checking where another fails. Unfortunately, developers make tool choices either in an ad-hoc manner or using rules of thumb such as selecting the tool that gives the best average runtime over a repository of software verification instances. As we shall show with concrete examples later, these may not necessarily be the optimal choices. While it might be feasible for a developer to evaluate all tools on a few instances relevant to her, in the industrial setting requiring deployment on large code bases, this would entail a huge cost, both in terms of time and money. The objective of this paper is to present an *algorithmic solution* for selection of optimal (in terms of runtime) and accurate (in terms of correctness of results) software model checkers.

Software model checking is an undecidable problem [43]. Nevertheless, the existing tools work successfully in practice on most, if not all, problem instances; but their performance is difficult to analyze theoretically. Besides, a lot of engineering optimizations [37] go into the design of these tools whose utility can only be understood by the tool designers. We therefore take a novel approach, based on machine learning, to synthesize an *algorithm selector* [39] for software model checkers. Our technique, called MUX, generates a *many-to-one selector* by training different machine learning algorithms on a repository of data obtained from runs of the tools. The synthesized algorithm selector $\mathcal{AS}$ uses structural features extracted from a program-property pair (*i.e.*, an input instance) at runtime and determines which tool to use.

Figure 1 shows the schematic view of MUX. The algorithm selector $\mathcal{AS}$ is synthesized offline by MUX in four steps. The *first step* involves running the available tools on a set of program-property pairs $(P_1, \varphi_1), \ldots, (P_n, \varphi_n)$ provided by the tool designers. Alternatively, the developers who use these tools could also supply them or these can be obtained from benchmarks used in software verification competitions [9]. The model checking problem being undecidable, the model checkers are only semi-algorithms and may not terminate in some cases. Therefore, MUX takes a timeout value Δ for running the tools. Software model checkers use complex data structures and

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597080

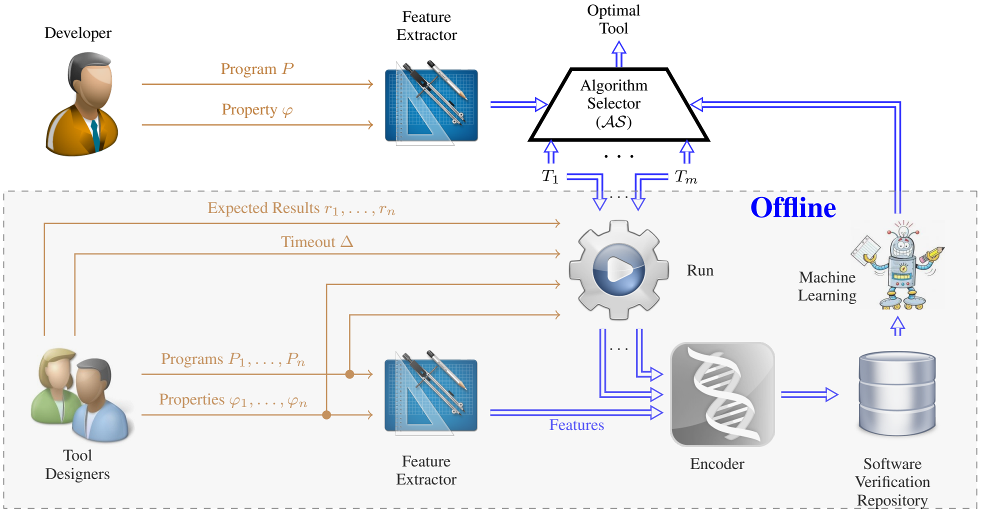

Figure 1: The schematic view of the Mux approach

algorithms, and may themselves contain faults, resulting in incorrect output in some cases. Mux therefore also takes the expected results $r_1, \dots, r_n$ as input where r_i indicates whether $P_i \models \varphi_i$ or not. If the tools are believed to be free of faults then this input can be omitted. The *second step* extracts features from each program-property pair. Mux extracts over a hundred features from the input instances. The features include statistics about the structure of programs such as the number of pointers, variables, functions, loops, and so on – some of which potentially affect tool performance. The properties to be model checked are typically given in the form of monitor functions encoding safety automata. Among other statistics, the number of states in these automata is also included as a feature. The feature vectors obtained from the input instances are encoded along with the timing and correctness results of the tools in the *third step* and are stored in a repository. In the *final step*, supervised machine learning algorithms are trained and the best machine-learned model among these is synthesized in the form of the algorithm selector $\mathcal{AS}$.

Mux offers the following key benefits compared to a manually-designed, heuristic-based algorithm selector: (1) In Mux, the mechanism for algorithm selection depends only on the features extracted from program-property pairs and is *independent* of internals of model checkers. Therefore, with suitable data, Mux can extend its set of choices with any model checker. Since it is difficult for humans to mine rules from large amounts of data, for manual design, this would require the developer to have a deep understanding of the tools. (2) Every time a new tool arrives or an existing tool improves its performance, manually updating the algorithm selector is not a viable option. In the fast moving field of software model checking, the algorithmic solution provided by Mux can continue to predict the most suitable tool from all the options available at any point of time – in a completely automated manner.

The notion of algorithm selection has been applied successfully in several other software engineering activities, *e.g.*, in compiler optimizations [44], code generation [27], and parallel programming [42]. In the verification domain, algorithm selection applied to satisfiability checking (SAT) produced impressive results, outperforming individual SAT solvers and winning competitions [47]. To the best of our knowledge, this paper is the first application of algorithm selection to software model checking. This work applies Mux to software model checkers for sequential C programs. Model checking is a popular technique with many tools available for concurrency model checking and bounded or symbolic model checking of software. Some benchmarks (*e.g.*, [1, 2]) are designed by the research community for evaluating them. With an appropriate selection of features, the Mux approach can potentially be applied to such model checkers as well.

To evaluate Mux, we have implemented the approach for Windows device drivers and safety properties, specified in the SLIC specification language [8], obtained from the Static Driver Verifier (SDV) framework. SDV is available as part of the Windows Driver Kit. There are currently three software model checkers available in SDV: SLAM [5], Yogi [38], and Corral [32]. These tools are based on different algorithmic techniques. More specifically, SLAM uses counter-example guided abstraction-refinement (CEGAR), Yogi combines static analysis and testing, whereas Corral performs bounded goal-directed symbolic search using SMT solvers. Since Mux does not require knowledge about internals of model checkers, these algorithmic variations pose no difficulty to Mux.

The feature extraction and algorithm selection, applied at runtime, add very little overhead but result in significant productivity gains. We experimented with 79 device drivers and the three model checkers. Each driver was model checked on all properties that were applicable from the set of 193 pre-defined properties in SDV. About half of the drivers were used for training and validation (*i.e.*, in the offline phase), and the other half for testing the performance of the algorithm selector $\mathcal{AS}$ synthesized by Mux.

Over the test instances, $\mathcal{AS}$ outperformed each of the individual tools. It reduced both the *number of timeouts* and the *total runtime* significantly. While SLAM results in timeout on 194 instances, Yogi and Corral result in timeout on 75 and 62 instances. $\mathcal{AS}$ performs the selection in a manner that results in only 37 instances timing out. Of these, all tools result in timeout on 16; this implies that $\mathcal{AS}$ makes only 21 incorrect choices (with respect to timeouts) over a few thousand test instances considered. Further, we compare the total time taken by $\mathcal{AS}$ selected tools and each of the individual tools on those instances where both finish within the timeout. On these instances, $\mathcal{AS}$ selected tools take less than 55% and 68% time respectively when compared to SLAM and Yogi. We note that each model checking run can be expensive, possibly running into hours. To put the productivity gain in perspective, while Yogi took over 4 days of CPU time, $\mathcal{AS}$ successfully completed the same tasks with *only* 2 *days and* 18 *hours* of CPU time.

Out of the instances available to us, there were 11 instances for which the output of some tool was incorrect. These cases arise from

Table 1: Average time taken by the tools on the benchmark programs: The optimal choices are underlined.

Property	SLAM	Yogi	Corral
φ_1	7657s	7192s	$\underline{\text{4577s}}$
φ_2	$\underline{\text{3508s}}$	6737s	3793s
φ_3	3422s	3647s	$\underline{\text{237s}}$
φ_4	3326s	4304s	$\underline{\text{387s}}$
φ_5	$\underline{\text{1649s}}$	3384s	2480s
Any	3912s	4407s	$\underline{\text{2295s}}$

implementation bugs in the tools. We trained MUX together on correct and incorrect instances with an appropriate encoding that differentiates between them. The algorithm selector obtained could avoid incorrect choices in the test instances in all but one case.

The algorithm selector $\mathcal{AS}$ is derived by MUX after extensive experimental evaluation of 25 machine learning problems which include a variety of classification and regression problems. In this, we also experimented with different encodings of the training instances. While in some encodings, all instances were treated as equal, in others the instances were weighted by the difference in the runtime of the best tool and the worst tool – thereby, assigning more importance to avoiding more expensive mistakes. Similarly, the machine learning algorithms were encouraged to avoid selecting a tool that could give an incorrect result. Non-linear support vector machine (SVM) classifier with weighted instances was the best performer over the validation data, and was synthesized as the algorithm selector $\mathcal{AS}$.

The SDV toolkit for Microsoft Windows uses a manually-designed *portfolio-based algorithm selector* [25] called Q. Over the test instances, Q could not prevent 51 timeouts, whereas the time taken by $\mathcal{AS}$ selected tools was only 78% of the time taken by the tools selected by Q. These results suggest that machine-learned algorithm selectors can outperform manually-designed algorithm selectors.

MUX performs feature selection before training the machine learning algorithms. In this, it discovers 14 features, from over a hundred features extracted from program-property pairs, that are highly-correlated with performance of the model checkers. Even though only empirically validated, these features may serve as a useful point of reference to both researchers and practitioners to study the factors affecting performance of the software model checkers.

To summarize, the main contributions of this paper are:

- A novel algorithmic solution to the problem of selecting software model checkers on per instance basis.
- Identification of key structural features of program-property pairs which are statistically correlated with the performance of software model checkers.
- An empirical training and evaluation of several machine learning algorithms resulting in automated synthesis of an algorithm selector that outperformed the individual tools and a manually-designed algorithm selector.

2. MOTIVATING EXAMPLE

With an example, we now present the challenges in selecting the optimal software model checker and motivate the MUX approach. We consider five properties that are pre-defined in SDV in this example. For brevity, these are denoted by φ_1 to φ_5. The data used in this example is taken from the actual repository.

One simple, and perhaps, common heuristic for algorithm selection is to select the tool that gives the *best average-case* performance on a *benchmark*. Table 1 gives the average time taken by each of the three model checkers – SLAM, Yogi, and Corral – for the five properties, φ_1 to φ_5, on some benchmark programs. It also gives the average time across all properties, tagged as "Any" in the property column. In some cases, the tools result in timeout on the benchmark programs. The timeout value here is set to 10K seconds. Timeouts increase the average time taken by a tool proportionately. For each row, the optimal choice is underlined. With the best average-case heuristic, for properties φ_2 and φ_5, SLAM is the best choice, whereas for the other three properties, Corral is the best choice. With this data, Yogi is not selected for any property. Once a tool is selected, the same tool is used for all input instances. This form of algorithm selection is therefore also called "winner-takes-all". We denote the algorithm selector induced by this mapping by AVG. If a tool is to be selected in property-agnostic manner then Corral will be selected, based on the last row in Table 1.

Table 2: Performance of the tools on some programs that were not in the set of benchmark programs: The optimal choices are underlined. TO indicates timeout.

Instance	SLAM	Yogi	Corral	Q	AVG	$\mathcal{AS}$
1 (P_1, φ_4)	2144s	1580s	$\underline{\text{1330s}}$	$\underline{\text{1330s}}$	$\underline{\text{C}}$	Y
2 (P_1, φ_5)	1650s	1560s	$\underline{\text{1534s}}$	2960s	S	Y
3 (P_2, φ_1)	TO	$\underline{\text{1222s}}$	TO	2622s	C[TO]	$\underline{\text{Y}}$
4 (P_2, φ_2)	TO	$\underline{\text{1648s}}$	2801s	3048s	S[TO]	$\underline{\text{Y}}$
5 (P_2, φ_4)	TO	TO	$\underline{\text{1367s}}$	$\underline{\text{1367s}}$	$\underline{\text{C}}$	Y[TO]
6 (P_2, φ_5)	TO	$\underline{\text{2416s}}$	2984s	3816s	S[TO]	C
7 (P_3, φ_1)	TO	TO	$\underline{\text{2416s}}$	TO	$\underline{\text{C}}$	$\underline{\text{C}}$
8 (P_3, φ_3)	TO	TO	$\underline{\text{2077s}}$	TO	$\underline{\text{C}}$	$\underline{\text{C}}$
9 (P_3, φ_5)	TO	TO	$\underline{\text{1114s}}$	$\underline{\text{1114s}}$	S[TO]	$\underline{\text{C}}$
10 (P_4, φ_1)	TO	$\underline{\text{1897s}}$	TO	3297s	C[TO]	$\underline{\text{Y}}$
11 (P_4, φ_2)	TO	TO	$\underline{\text{2894s}}$	TO	S[TO]	Y[TO]
12 (P_4, φ_3)	2999s	TO	$\underline{\text{2942s}}$	TO	$\underline{\text{C}}$	$\underline{\text{C}}$
13 (P_4, φ_5)	TO	TO	$\underline{\text{832s}}$	$\underline{\text{832s}}$	S[TO]	$\underline{\text{C}}$
14 (P_5, φ_4)	TO	TO	$\underline{\text{1107s}}$	$\underline{\text{1107s}}$	$\underline{\text{C}}$	$\underline{\text{C}}$
15 (P_6, φ_2)	TO	$\underline{\text{2046s}}$	TO	3446s	S[TO]	$\underline{\text{Y}}$
16 (P_6, φ_3)	TO	$\underline{\text{2771s}}$	TO	4171s	C[TO]	$\underline{\text{Y}}$
17 (P_6, φ_4)	TO	$\underline{\text{1484s}}$	TO	2884s	C[TO]	$\underline{\text{Y}}$
Timeouts	14	8	5	4	10	2

The algorithm selector used by SDV is called Q. Technically, it is a portfolio-based algorithm selector. A *portfolio-based algorithm selector* maintains a set of tools and runs one or more of them on an input instance, either parallelly or in a time-sliced manner [25]. In particular, out of the three model checkers, Q utilizes only Yogi and Corral in the portfolio and runs Corral initially up to 1400s and then runs Yogi with a higher timeout value. The design of Q is based on two key observations: (1) Corral, when works, finishes fast. The timeout of 1400s is obtained as a lower bound on the runtime of instances on which Corral takes more time than Yogi. Thus, if Corral does not finish in 1400s then switching to Yogi is beneficial. (2) Corral and Yogi mostly result in timeout on disjoint instances but Yogi takes more time on an average for instances on which both succeed. That is why Corral is run first. Note that the criterion used in Q is completely agnostic to the input instances.

Table 2 gives the runtimes of the tools on some programs that were *not* part of the benchmark programs. The properties in these instances are from the properties listed in Table 1. TO indicates timeout. The optimal choice for each instance is underlined. In these 17 instances, SLAM times out on 14, Yogi and Corral result in timeout on 8 and 5 instances respectively. Note that the performance of the property-agnostic variant of AVG, discussed above, coincides with that of Corral. We now analyze the performance of AVG and Q on the instances in Table 2. For the algorithm selectors, the choices are abbreviated with the first letter of the tool name. AVG makes 6 optimal choices, only 1 sub-optimal choice (that succeeds without timing out), but 10 choices that result in timeouts (worse than any individual tool). On the other hand, Q makes 5 optimal and 8 sub-optimal choices, and times out on 4 instances.

The developer might be tempted to tweak the criteria discussed above by observing some patterns in the data. For instance, if Q sticks with Corral for longer then it can avoid all the 4 timeouts – instances 7, 8, 11, and 12. However, this will penalize Q's performance on several other instances where Yogi must be used anyway. Instead of manually exploring such ad-hoc selection criteria, we propose an *algorithmic approach* in this work.

The key limitation of the strategies discussed above is that they are too coarse-grained. The performance of a model checker, or any algorithm for that matter, is a function of all its inputs. However, neither AVG nor Q analyze the input instances while selecting a tool – they only rely on an approximate characterization of the past performance of the tools. In contrast, our approach, called MUX, uses structural features of input instances to learn and then predict the best tool on *each individual input instance*. Similar to AVG and Q, MUX also uses the past performance of the tools but tries to learn a more accurate prediction model.

Table 2 shows the choices made by the algorithm selector $\mathcal{AS}$ synthesized by MUX (see the last column). $\mathcal{AS}$ makes optimal choices on 12 instances and sub-optimal choices on 3 instances. It results in timeout only in 2 cases which is less than timeouts for any individual tool or the manually designed algorithm selectors.

Apart from timeouts, the total time taken to model check all the instances is also an important metric affecting developer productivity. There are 12 instances on which neither $\mathcal{AS}$ nor Q timeout. On these instances, the tools selected by $\mathcal{AS}$ take only 66% of the time taken by the tools selected by Q. This is because whenever Q makes a sub-optimal choice, its performance degrades sharply. In fact, it takes *more time than any individual tool*. This is a consequence of its time-slicing based scheduling which would run both Corral and Yogi in such cases. In contrast, $\mathcal{AS}$ is a many-to-*one* selector and hence, does not suffer overheads of running more than one tool.

3. SYNTHESIS OF ALGORITHM SELECTOR

In this section, we present the design steps of our approach and the algorithm MUX for synthesis of an algorithm selector.

3.1 Design Steps

3.1.1 *Feature Identification*

In machine learning, the input instances are modeled as values of a finite set of features. A *feature* is a numerical, efficiently computable property of the input instance. For a machine learning algorithm to learn good prediction models, the set of features should characterize the input instances appropriately and at least some of the features must correlate well with the expected output. Further, the feature computation must be efficient. This requirement is even more important in our setting. To obtain real benefits from algorithm selection, the overheads of feature computation (and tool prediction) should only be a small fraction of the time required for model checking the corresponding instances.

MUX computes over a hundred *structural features* of a given verification instance (*i.e.*, a program-property pair). Table 3 gives a representative list of features. These include program-based features like count of *scalar variables* of a primitive type such as integers, count of *void pointers*, count of different *types of statements* in the program, and so on. Besides, features about properties such as its *type* (*e.g.*, bit-vector, safety property) and *count of states in the safety automaton* represented by the property are included. These features are computed efficiently with a single pass over the entire program.

We give insights about how some of these features *may* affect runtime of the model checkers. Corral [32] performs abstractions only over global variables of a program, while maintaining local variables explicitly. MUX separately tracks the count of variables, pointers, etc. by *lexicographic scope* (see Table 3). These features may help differentiate programs on which Corral performs well from those on which it does not. The abstraction computed by SLAM [7] is exponential in the number of predicates. The initial set of predicates used for refinement corresponds to the predicates in conditionals of the program. Thus, *count of conditional statements* can play an important role in predicting its performance. Pointers and function calls are handled differently in different model checkers. For instance, Yogi [38] computes aliasing information only during symbolic execution and may scale better on programs with many pointers.

Table 3: A representative list of features

Feature type	*Example features*
Features extracted from programs	
Scalar variables	Count based on types *e.g.*, integers, floats, characters
Arrays	Count based on types *e.g.*, integer arrays, character arrays
Aggregate datatypes	Count of structs and unions
Pointers	Count based on types *e.g.*, integer ptrs, function ptrs, void ptrs
Lexicographic scope (applicable to all the feature types above)	Separate counts for local and global variables
Expressions	Count of arithmetic expr., array indexing, pointer dereferences
Statements	Count of assignments, conditionals, loops, function calls
Functions	Count of recursive functions, Avg. number of func. arguments
Complexity measure	Cyclomatic complexity
Features extracted from properties	
Property type	Bit-vector, Safety property
Annotations	Count of watch/guard constructs, Number of state variables
States of automaton	Count of entry/exit/total states

While it might be possible to guess some features (such as above) which could be useful for predicting performance of certain tools, in designing MUX, we have tried to include as many features as possible. Thus, our design is *not* specific to the three model checkers considered in this paper. Having more (and even spurious) features does not affect the accuracy of MUX. Similar to most of the machine learning approaches, MUX does *feature selection* to identify statistically important features as a pre-processing step. In our experiments, finally only 14 features were used in prediction. We discuss the results of feature selection in Section 4.2.

3.1.2 *Formulation as Machine Learning Problems*

In the case of algorithm selection, since the best tool is known for the training instances, it is natural to formulate it as a *supervised learning* problem. In supervised learning, the training data is labeled with desired outcome. Classification and regression are the two predominant classes of supervised learning problems [13].

The algorithm selection problem can be formulated as a *multi-class classification* problem as follows: Given a program-property pair (P, φ) determine its label – where *label* ranges over identifiers of the model checking tools. Unlike binary classification which considers only two labels, multi-class classification works over any finite set of labels. The labeling results in a *mis-classification* if the tool identified by the classification model (*i.e.*, model learned by the classification learning algorithm) does not give the lowest runtime among all the available tools. When a tool produces an incorrect re-

Table 4: Machine learning problems solved in MUX

Machine learning algorithm	ML problems
Non-linear SVM without weights	3 binary, 1 multi-class
Non-linear SVM with weights	3 binary, 1 multi-class
Non-linear SVM with discretized weights	3 binary, 1 multi-class
Linear SVM without weights	3 binary, 1 multi-class
Linear SVM with weights	3 binary, 1 multi-class
Linear SVM with discretized weights	3 binary, 1 multi-class
Linear regression	Ridge regression
Total number of problems	25

sult (due to a fault), its runtime is set to a constant higher than the timeout value. This discourages the machine learning algorithms from selecting the tool for other instances where the tool is likely to produce incorrect results. This also provides them with a uniform, quantitative criterion (runtime) to optimize.

Considering all the tools together as part of a classification problem may affect accuracy of prediction unnecessarily, if a tool is not suitable in most cases. To overcome such a possibility, we consider a more fine-grained classification problem, namely, multiple instances of *binary classification* – where we consider every pair of tools separately. The notion of mis-classification, defined above for multi-class classification, remains the same but is restricted to only the two tools being considered.

In the setting of algorithm selection, for each input instance we can try to predict the runtime of each tool. This version can be formulated as a *regression* problem: Given a program-property pair (P, φ), predict the runtime of each tool. The goal of regression is to learn a regression model, also called an estimator, of the actual function which in our case determines the runtime. Once we obtain regression models for all the tools, on every input instance, the tool which has the lowest predicted runtime is selected.

3.1.3 Definitions of Objective Functions

The features identified in Section 3.1.1 define the *input* to a machine learning problem. We now formalize the *output*.

The algorithms to solve classification and regression problems have well-defined objective functions. In classification, the objective is to minimize the mis-classifications. In regression, it is to minimize the mean squared error or some variant of it (*e.g.*, root mean squared error) between the function and its estimation computed by the algorithm. It is also common to consider *weighted instances* in classification where a mis-classification penalty is scaled by the weight of the instance being mis-classified. In our context, we derive weights by considering the runtimes of the tools. In particular, apart from unweighted instances, MUX encodes two types of weighted instances, where (1) weight is equal to the *difference in runtime* between the best and the worst tool and (2) weight is equal to *discretization level of the difference in runtime* between the best and the worst tool. For the latter, we select a discretization value, say δ. If d is the difference in runtime then the discretized weight is k such that $(k-1).\delta < d \leq k.\delta$.

Clearly, the weight indicates importance of an instance whereas without weights, all instances are equal. It is undesirable to select a sub-optimal tool when the difference in runtimes is high. Conversely, when the difference is low, a sub-optimal choice is acceptible, providing more flexibility in machine learning. The weight is likely to be high for instances where (1) one of the tools times out and another does not, or (2) one of the tools produces an incorrect result and another does not. As we discuss in Section 4, a classification algorithm *with* weights gave the best results in our experiments.

The classification and regression problems can be solved with linear as well as non-linear techniques. Table 4 gives the list of machine learning algorithms that were evaluated as part of MUX. In the table, the weight measure that uses difference in runtime of the best and the worst tool is denoted simply as "with weights". The discretized variant of it is denoted as "with discretized weights". The first six rows correspond to different classification algorithms. For each of them, MUX solves 4 classification problems: 3 of them are binary classification problems (over the pairs of the three software model checkers considered) and 1 is a multi-class classification problem (over all three model checkers). The multi-class classification problem is solved by taking majority vote among the binary classifiers. In our setting, in the case of a tie, we want to select a specific model checker (Corral). Existing approaches for generating multi-class classifiers may break the ties arbitrarily and hence, we do not use them. The non-linear support vector machine (SVM) algorithms are used with Gaussian kernel. In all, MUX solves 25 machine learning problems and selects the best solution.

Algorithm 1: Algorithm MUX

```
   Input: Program-property pairs $(P_1, \varphi_1), \ldots, (P_n, \varphi_n)$, the
          corresponding expected results $r_1, \ldots, r_n$, model
          checkers $T_1, \ldots, T_m$, and timeout $\Delta$
   Output: An algorithm selector $\mathcal{AS}$
 1 begin
 2     $DB \leftarrow \emptyset$ // Initialize the repository
 3     foreach program-property pair $(P_i, \varphi_i)$ do
 4         foreach model checker $T_j$ do
 5             $Time_{ij} \leftarrow$ Run $T_j$ up to time $\Delta$ to check $P_i \models \varphi_i$
 6             if $Time_{ij} < \Delta$ then
 7                 Let $Result_{ij}$ be the result of $T_j$ on $(P_i, \varphi_i)$
 8                 if $Result_{ij} \neq r_i$ then $Time_{ij} \leftarrow 2 \times \Delta$
 9         end
10         $FV_i \leftarrow$ Extract structural features from $(P_i, \varphi_i)$
11         $DB \leftarrow DB \cup \{\langle FV_i, Time_{i1}, \ldots, Time_{im} \rangle\}$
12     end
       // Select training and validation sets
13     $TS \leftarrow$ Select a subset of $\{(P_1, \varphi_1), \ldots, (P_n, \varphi_n)\}$
14     $VS \leftarrow \{(P_1, \varphi_1), \ldots, (P_n, \varphi_n)\} \setminus TS$
       // Perform feature selection
15     $\mathcal{F} \leftarrow$ Select a subset of most useful features using $TS$
       // Train ML models
16     Train classification models on $DB$ restricted to $TS$ and $\mathcal{F}$
17     Train regression models on $DB$ restricted to $TS$ and $\mathcal{F}$
       // Select the best ML model
18     $C \leftarrow$ Select the best classification model wrt $VS$
19     $R \leftarrow$ Select the best regression model wrt $VS$
20     $B \leftarrow$ Select the best model between $C$ and $R$
       // Synthesize the algorithm selector AS
21     $\mathcal{AS} \leftarrow$ Synthesize $B$ as a program from feature vectors to
       identifiers of the model checking tools
22 end
```

3.2 Algorithm

The algorithm MUX (see Algorithm 1) takes as input a set of program-property pairs $(P_1, \varphi_1), \ldots, (P_n, \varphi_n)$, the expected results $r_1, \ldots, r_n$ where r_i indicates whether $P_i \models \varphi_i$ or not, a set of model checking tools $T_1, \ldots, T_m$, and a timeout value Δ. The output of MUX is an algorithm selector $\mathcal{AS}$: a program to map feature values of a program-property pair to a tool identifier.

MUX uses a repository DB to store the information about each model checking run. It initializes DB to the empty set (line 2). The algorithm then runs each available tool T_j on every program-property pair (P_i, φ_i) to obtain the time taken by the tool for checking whether $P_i \models \varphi_i$ (line 5). Let $Time_{ij}$ be the time taken by T_j

on (P_i, φ_i). If T_j does not terminate before the timeout occurs then MUX aborts the model checking run, setting $Time_{ij}$ to the timeout value Δ. If T_j finishes before the timeout (line 6) but its result is incorrect (line 8) then MUX resets $Time_{ij}$ to a value higher than Δ. In this work, we set it to $2 \times \Delta$ but this is configurable. MUX stores the feature vector FV_i for (P_i, φ_i) along with runtimes of all the tools in the repository (lines 10 and 11).

The next step partitions the data into training set TS and validation set VS (lines 13 and 14). The training data TS is used for training the machine learning algorithms and the validation data VS is used for comparing their performance in order to identify the best model to be synthesized as the algorithm selector.

In a fine-grained partitioning, the individual program-property pairs can be partitioned into two sets. With such a partitioning, for the same program P_i, data on some properties may end up in the training set and data on other properties in the validation set. For instance, consider a program-property pair (P_1, φ_1) belonging to the validation set such that some (P_1, φ'_1) is present in the training set. We note that the set of features pre-dominantly include features about programs (see Section 3.1). During validation, a machine learned model may give an accurate result on (P_1, φ_1) due to its similarity to (P_1, φ'_1). Due to this, with the fine-grained partitioning, the machine learned model selected by MUX could be the one that is *over-fitted* to the training data. Consequently, it is more likely to fail (in selecting the optimal, accurate model checker) on an instance (P_2, φ_2) such that there is no instance (P_2, φ'_2) in the training set. MUX avoids such a situation by *partitioning programs* (instead of program-propery pairs) into two sets A and B and then puts all program-property pairs (P_i, φ_i) into the training set TS if $P_i \in A$. Analogously, all program-property pairs (P_j, φ_j) such that $P_j \in B$ are included the validation set VS.

To start with, the features that are important for good prediction are not known. MUX computes all features and then selects a promising subset using a suitable feature selection mechanism (line 15). More specifically, it computes Pearson's correlation coefficient [29] of every feature with the runtime of each of the tools. Pearson's correlation coefficient is the numerical measure of the strength of linear correlation between two statistical variables. The coefficient value close to 1 indicates that the two variables have a strong linear dependence on each other. It then selects the features whose correlation coefficient is above a threshold (which is 0.35 in the current implementation) for at least one tool. Let $\mathcal{F}$ be the set of features selected. The subsequent training and validation steps are performed only with respect to these features.

MUX now trains the classification and regression algorithms on the training data TS (lines 16 and 17). For multi-class classification, MUX encodes a training instance $\langle FV_i, Time_{i1}, \ldots, Time_{im} \rangle$ as a triple $\langle \mathcal{FV}_i, \ell_i, w_i \rangle$ where $\mathcal{FV}_i$ is the feature vector restricted to the features from the set $\mathcal{F}$, ℓ_i is the expected label, and w_i is the weight. MUX computes the label ℓ_i as $\ell_i = \arg\min_j \ Time_{ij}$ for $j \in \{T_1, \ldots, T_m\}$. Let

$$\begin{aligned} max_i &= \max\{Time_{i1}, \ldots, Time_{im}\} \text{ and} \\ min_i &= \min\{Time_{i1}, \ldots, Time_{im}\}. \end{aligned}$$

The weight w_i can take one of the following values depending on whether weighted/unweighted instances are used:

$w_i = \kappa$	unweighted instances
$w_i = max_i - min_i$	diff. in runtime of best/worst tools
$w_i = \lceil (max_i - min_i)/\delta \rceil$	discretized difference in runtime

The constant δ is the discretization value and κ is an arbitrary positive constant. The aim of machine learning in this case is to derive a function from feature vectors to the set of labels $\{T_1, \ldots, T_m\}$.

For binary classification, from one training instance, multiple encodings – one each for every pair of tools – are created. An individual encoding is analogous to the encoding for multi-class classification but restricted to only the pair of tools being considered. In the case of regression, MUX generates one encoding each for every tool. The encoding of an instance $\langle FV_i, Time_{i1}, \ldots, Time_{im} \rangle$ for a tool T_j is simply $\langle \mathcal{FV}_i, Time_{ij} \rangle$ and the aim of (linear) regression learning is to compute an (linear) estimator of the function from the feature vectors to the runtime of the tool. The objective functions for all the machine learning problems, with and without weights, are defined in Section 3.1.3.

MUX identifies the best classification model, from the trained ones, on the validation set VS (line 18). Since some classifiers do not use weights and others use one of the two notions of weights, to compare them uniformly, MUX assigns the *score* for a prediction as runtime of the predicted tool. Given the feature vector $\mathcal{FV}_i$ of an instance $\imath = \langle FV_i, Time_{i1}, \ldots, Time_{im} \rangle$, if a classification model selects label ℓ then the score is $S(\imath, \ell) = Time_{i\ell}$. The *aggregate score* of a classification model $\mathcal{C}$ over the validation set is equal to $\sum_{\imath \in VS} S(\imath, \mathcal{C}(\mathcal{FV}_i))$. Naturally, a mis-classification results in a higher increment of the aggregate score. The best classification model is the one with the least aggregate score.

Note that the criterion, defined above, for identification of the best classification model differs from the usual notions of accuracy and precision [13]. In our setting, the total time taken by the predicted tools is more important that the individual predictions. For instance, the tools predicted by a classification model $\mathcal{C}_1$ with 90% precision may take more time to finish the tasks than those predicted by another classification model $\mathcal{C}_2$ with, say only 80% precision. In such a case, MUX prefers $\mathcal{C}_2$ over $\mathcal{C}_1$.

As discussed in Section 3.1, MUX trains one regression model for each of the tools. The prediction of the combined regression model R on an instance $\imath$ is the tool with smallest predicted runtime. Suppose ℓ is the tool predicted by R. The definitions of score $S(\imath, \ell)$ and aggregate score here are the same as the definitions in the classification case. MUX then selects the best model, by aggregate score, between the best classification model and the best regression model (line 20). Finally, the algorithm selector $\mathcal{AS}$ is synthesized. It is a straightforward encoding of the function defined by the best machine-learned model.

4. EVALUATION

We use Static Driver Verifier (SDV) as our framework for implementing MUX. The feature extraction is implemented in OCaml, as an extension of the SDV compiler. Feature selection is implemented with Weka [23] and the machine learning algorithms are implemented using Matlab[1], LibLinear [17], and LIBSVM [15].

Research Questions. We evaluate the following questions about the design choices made by MUX during the **offline phase**:

(RQ1) *Selection of important features* – Which features are identified by MUX as the most important ones for predicting the optimal and accurate software model checkers?

(RQ2) *Choice of the machine learned model* – Which machine learned model is used by MUX as the algorithm selector $\mathcal{AS}$?

(RQ3) *Performance and scalability* – How does MUX perform in the training and validation steps?

Next, we evaluate effectiveness of the algorithm selector $\mathcal{AS}$ synthesized by MUX in the **online phase**:

(RQ4) *Reduction in the number of timeouts* – Does the algorithm selector $\mathcal{AS}$ reduce the number of timeouts, compared to the individual tools and Q?

[1] http://www.mathworks.com/products/matlab/

Table 5: Features identified by Mux as important for algorithm selection and their correlation coefficients

	Feature description	*Corr. coefficients*		
		SLAM	*Yogi*	*Corral*
1	Total number of local variables	0.45	0.39	0.46
2	Total number of conditional statements	0.39	0.31	0.31
3	Total number of formals and local variables	0.45	0.40	0.47
4	Total number of function calls	0.45	0.27	0.47
5	Total number of local integer-type variables	0.46	0.39	0.46
6	Total number of local scalar variables	0.45	0.29	0.45
7	Total number of local struct or union variables	0.52	0.39	0.50
8	Total number of local pointers	0.46	0.31	0.48
9	Total number of local pointers to struct or union	0.46	0.42	0.49
10	Total number of local void pointers	0.46	0.30	0.34
11	Total number of local function pointers	0.45	0.45	0.45
12	Is the driver implemented using C++ (or only C)?	0.07	0.13	0.08
13	Does the property utilize bit-vector operations?	0.03	0.04	0.02
14	The number of states of the property automaton	0.10	0.26	0.18

(RQ5) *Productivity gain* – Does the algorithm selector $\mathcal{AS}$ improve the total time required for model checking large code bases?

(RQ6) *Performance and scalability* – Does the algorithm selector $\mathcal{AS}$ add runtime overheads in the online phase?

4.1 Experimental Setup

For our experiments, we chose the test suites that are available in the SDV framework. Specifically, due to its inherent complexity, we use the Windows Driver Model (WDM) driver test suite, which contains 79 drivers, developed independently by different teams. This test suite consists of production as well as test drivers that range from 1K to 50K LOC in size. For the WDM drivers, SDV contains a total of 193 properties. However, not all properties are applicable to each driver. Using the above test suite, we get a total of 5717 program-property pairs for evaluation. Thus, Mux is evaluated on a large, industrial code base and a large number of verification instances. In our experiments and data collection, we used a machine with two Intel Xeon processors (16 logical cores) executing at 2.4 GHz and a total of 32 GB RAM. The model checkers are sequential and analyze an instance only on a single core.

The usual practice in machine learning is to train the algorithms with examples that are reflective of the test data. In our case, we need examples of drivers for which each model checker gives the least total runtime over all the properties; otherwise, the training data would be under-representative of the test data. We therefore partitioned the drivers according to the best model checker. The drivers from each set were then classified randomly into training, validation, and test data. More specifically, only 35% drivers (and all their properties) were used for training, whereas 15% drivers were used for validation. The remaining 50% drivers were presented to the algorithm selector $\mathcal{AS}$ only at runtime. In terms of program-property pairs, 2797 pairs were used in the offline phase (for training plus validation) and the remaining 2920 pairs were used in the online phase (for testing). We call this setup *baseline setup* and evaluate Mux in Sections 4.2 and 4.3 using it.

In this repository, there were additional 11 instances for which the output of some tool was *incorrect*. We evaluate Mux by adding these instances to the baseline setup in Section 4.4.

4.2 Evaluation of the Offline Phase

We now evaluate the offline phase in which Mux solves the machine learning problems (see Table 4) in order to identify and synthesize the best algorithm selector.

(RQ1) Selection of important features

Mux starts with a feature set of 131 structural features over the program-property pairs and identifies the most important features through statistical analysis as discussed in Section 3.2. Mux discovers that only a few features shown in Table 5, are important for prediction. In the remaining steps, Mux performs machine learning with respect to only these features.

Table 6: Comparison of machine learning algorithms

Machine learning algorithm	Time	Timeouts
Non-linear SVM without weights	19760s	0
Non-linear SVM with weights	19510s	0
Non-linear SVM with discretized weights	19588s	0
Linear SVM without weights	42016s	2
Linear SVM with weights	53163s	3
Linear SVM with discretized weights	41744s	2
Linear regression	44259s	2

Against each feature, Table 5 indicates the correlation coefficients of the feature with the runtime of each model checker. The first 11 features are selected by Mux and are all related to programs. We therefore manually added the last three features. Among these, the last two features are about the properties being model checked. Without these features, the instances of the same program with different properties would be mapped to the same feature vector, potentially affecting the machine learning accuracy adversely. The other manually-added feature indicates whether C++ constructs are involved since the tools handle them differently.

Interestingly, several features have high correlation with runtimes of each of the model checkers. As remarked in Section 3.1, it is known that the performance of SLAM gets affected by increase in the number of conditional statements in the program, while the same is not necessarily as critical for Yogi and Corral. The feature at position 2 in Table 5 captures this structural property of programs and its correlation with SLAM is higher than the other tools. Another interesting observation relates to the feature at position 4 which is the count of the total number of funtion calls in the program. Corral works by inlining function calls on demand. Clearly, the number of function calls has a relevance to the performance of Corral and Mux could identify this algorithmically, through statistical analysis. Even though only empirically validated, we believe that the features identified by Mux can serve as a useful reference to researchers and practitioners interested in design of efficient model checkers.

(RQ2) Choice of the machine learned model

Mux trains multiple machine learning algorithms on the training data. Each of these algorithms takes one or more hyper-parameters. The hyper-parameters capture the prior distribution from which the training data is presumably sampled [13]. A right configuration of hyper-parameters improves the performance of the machine learning algorithm. Mux obtains candidate configurations of hyper-parameters by uniformly sampling values from some ranges supplied by us. This approach is referred to as *grid search*.

In all, 25 machine learning problems (see Table 4) are solved by Mux. Each machine learned model is then evaluated on the validation set. Table 6 gives the runtime of the tools predicted by each algorithm and the number of timeouts that would result from their predictions. Similar to Table 4, the first six rows correspond to classification algorithms and the last one corresponds to a regression algorithm. We recall that Mux trains 3 binary classifiers and 1 multi-class classifier using each of the classification algorithms (see Section 3.1). The entries in the first six rows correspond to the best classifier among the four classifiers trained by the same algorithm. The *optimal* choices would have resulted in 0 timeouts and would take 16095s for model checking the instances in the validation set. Mux chooses the model which gives the least time on the validation set, called the *aggregate score* in Section 3.2. The minimal runtime is underlined in Table 6. It corresponds to *non-linear support vector machine (SVM)* with the difference in runtime of the best and the worst tool as the *weight measure*. Further, the classifier model (from among the four classifiers trained through the same algorithm) corresponds to a *binary classifier between Yogi and Corral*. This classifier is synthesized by Mux as the algorithm selector $\mathcal{AS}$.

Interestingly, Yogi and Corral are precisely the model checkers used by Q also. However, unlike Q which tries these tools in a certain order (Corral followed by Yogi), $\mathcal{AS}$ predicts one of them algorithmically. Besides, as our results on the test instances would show, $\mathcal{AS}$ outperforms the hard-coded logic of Q.

(RQ3) Performance and scalability

Mux finishes the entire training and validation tasks within a few minutes overall (the maximum is 7m). This includes the time taken for learning models with different configurations of hyper-parameters and in the case of the individual classification algorithms, for all the four classifiers together. In the case of linear SVM, Mux tried about 50 hyper-parameter configurations, whereas in all other cases, these were about 200. Thus, the training and validation steps of Mux scale very well and can efficiently identify the best machine learning model from a large set of candidates.

Before training, for each software verification instance, we run all the software model checkers to obtain respective timing and correctness results. For the purposes of our experiments, we divided the available programs equally between the offline and online phases. In practice, we expect a much larger number of programs being model checked online compared to the offline phase, offsetting the cost of collecting the training data.

4.3 Evaluation of the Online Phase

We now evaluate the performance of the *algorithm selector* $\mathcal{AS}$ synthesized by Mux on the test data.

(RQ4) Reduction in the number of timeouts

Table 7 gives the timeouts on the test data for the individual tools as well as the algorithm selectors $\mathcal{AS}$ and Q. The row/column numbers are indicated against the tool names. For simplicity, we refer to a tool or an algorithm selector by its row/column number. An (i, j)th entry in the table gives the number of program-property pairs on which both the tool (or algorithm selector) labeling the ith row and the tool labeling the jth column timeout. A diagonal entry (i, i) gives the total timeouts for the ith tool. The matrix is symmetric and hence only values in the lower triangle are given. The column labeled "optimal" corresponds the actual optimal choices for the test data. There are 16 instances on which all the model checkers timeout. This is the number of timeouts corresponding to the optimal choices, as these instances result in timeout irrespective of the choice of the model checker. The timeout value in the experiments was set to a substantially high value, 10K seconds, nearly 3 hours.

Among the software model checkers, Corral times out on the smallest number of instances (62), whereas SLAM times out on the maximum number (194). In contrast, the $\mathcal{AS}$ selected tools timeout only on 37 instances. As discussed above, 16 of them timeout with every tool. Thus, $\mathcal{AS}$ makes only 21 choices in which the $\mathcal{AS}$ selected tool times out but there is another tool which could have model checked that instance within the timeout value. The manually-designed algorithm selector Q, used in SDV currently, times out on a larger number of instances (51) compared to $\mathcal{AS}$.

For a pair of distinct tools i and j, the difference between (i, i)th and (i, j)th entries (for $j < i$) gives the number of instances on which the ith tool times out but the jth tool does not. Analogously, the difference between (j, j)th and (i, j)th entries (for $j < i$) gives the number of instances on which the jth tool times out but the ith tool does not. For example, between SLAM ($i = 6$) and $\mathcal{AS}$ ($j = 2$), SLAM times out on 166 instances on which the tools selected by $\mathcal{AS}$ do not timeout, whereas the tools selected by $\mathcal{AS}$ timeout on 9 instances on which SLAM does not timeout. Thus, with respect to SLAM, $\mathcal{AS}$ avoids timeouts on a *larger set* of instances while newly inducing timeouts only on a much smaller set of instances. This holds true for the other two model checkers as well as the algorithm selector Q. For instance, there are 10 instances on which $\mathcal{AS}$ results in timeout but Q does not and 24 instances where Q results in timeout but $\mathcal{AS}$ does not. Overall, $\mathcal{AS}$ successfully eliminates a significant number of timeouts originating from each of the model checkers and the choices of Q.

Table 7: Pairwise comparison of timeouts

	(1) *Optimal*	(2) $\mathcal{AS}$	(3) *Q*	(4) *Corral*	(5) *Yogi*	(6) *SLAM*
(1) *Optimal*	**16**					
(2) $\mathcal{AS}$	16	**37**				
(3) *Q*	16	27	**51**			
(4) *Corral*	16	23	27	**62**		
(5) *Yogi*	16	31	35	17	**75**	
(6) *SLAM*	16	28	45	50	42	**194**

Table 8: Comparison of the total runtime

Tool	*Instances*	*Time*	$\mathcal{AS}$
Optimal	2883	2d 09h	3d 03h (131%)
Q	2859	3d 05h	2d 12h (78%)
Corral	2844	2d 19h	2d 15h (94%)
Yogi	2839	4d 04h	2d 18h (68%)
SLAM	2717	3d 06h	1d 19h (55%)

(RQ5) Productivity gain

We want to now compare the total time taken by the tools selected by $\mathcal{AS}$ and each of the model checkers to model check the instances from the test data, *excluding* the instances on which either of them times out. Table 8 shows the number of instances on which the tool and $\mathcal{AS}$'s choices both finish within the timeout. As can be seen, the time taken by $\mathcal{AS}$'s choices is far lower than the time taken by Yogi or SLAM. It is only 68% and 55% respectively. Since the drivers considered in these experiments are fairly large, the time required for model checking a single program-property pair is usually high – of the order of several minutes or more. The model checkers required several days to complete model checking for all program-properties in the test data. For example, Yogi took over 4 days. A 32% saving on such a large CPU time is therefore quite significant. In particular, $\mathcal{AS}$ saves at least *one full day* as compared to Yogi. The same is true in the case of SLAM as well.

The time taken by Corral is reasonably close to that of $\mathcal{AS}$'s choices. We highlight that $\mathcal{AS}$ does *not* degenerate into selecting Corral by default. $\mathcal{AS}$ selects Corral only on 44% instances in the test data. Further, as discusssed earlier, the number of timeouts that Corral results in is much higher compared to that of $\mathcal{AS}$. Q reduces the number of timeouts compared to any individual model checker (though still higher than $\mathcal{AS}$). Unfortunately, it does not show any improvement with respect to the total runtime. As seen from Table 8, $\mathcal{AS}$'s choices take only 78% time compared to Q's choices. At this point, the performance of $\mathcal{AS}$ is somewhat farther from the optimal time. It is 31% slower than the algorithm selector which always selects the optimal tool.

(RQ6) Performance and scalability

The features are extracted by performing a *single* pass over the program and the description of the property being model checked. Therefore, unsurprisingly, the runtime overhead of $\mathcal{AS}$ in the online phase is very small. On the test data, $\mathcal{AS}$ typically takes a few seconds to extract features and predict the best tool. Even on the largest driver (50KLOC), the overhead was less than 10s.

4.4 Evaluation with Incorrect Instances

We now evaluate MUX by adding to the baseline setup the instances on which some tool produced an incorrect result. Out of the 11 such instances, 7 fell into the training set and 4 in the test set. As before, MUX selected a binary classifier between Yogi and Corral as the algorithm selector $\mathcal{AS}$. $\mathcal{AS}$ chose the tool that gave an incorrect result in 1 instance in the online phase. In comparison, Q chose incorrect tools in 2 cases. The choices of the algorithm selector $\mathcal{AS}$ caused 44 timeouts, better than individual tools and Q. For the non-timeout cases, $\mathcal{AS}$ continued to outperform the other tools and Q, with the percentage gain in total runtime about the same as Table 8. As discussed in Section 3.2, the runtime in the case of incorrect instances is set to twice the timeout value. Since the repository available to us at present does not have enough incorrect instances for further experimentation, we leave it to future work to evaluate whether varying this encoding can reduce the timeouts without increasing the incorrect choices or vice versa.

4.5 Threats to Validity

The threats to *internal validity* include selection bias where the training data and test data may have an overlap, leading to falsely pronounced results in the machine learning accuracy on the test data. We mitigated this threat by partitioning the data not by program-property pairs but by programs (see Sections 3.2 and 4.1). Thus, no program-property pair occurs in the two sets, and even more strongly, no program occurs in the two sets. We note that only 11 instances were identified as exposing implementation bugs in the tools. This was based on the explicit tagging by the developers who wrote the device drivers. The tool designers have confirmed them. However, it is difficult for us to ascertain that there are no additional instances with incorrect results. Another threat to internal validity may arise due to software bugs. We tested our implementation extensively and believe that there are no bugs in it.

Threats to *external validatity* arise because our results may not generalize to other classes of programs or properties and software model checkers. The device drivers considered in this work come from an industrial framework consisting of both production and test drivers, developed by different teams. These can therefore be considered as representative of device drivers in general. This however mitigates the threat only to an extent. It is not clear whether MUX can produce similar results on programs other than device drivers. A large number of properties were part of the data used in this work. From our experience in software model checking, we believe that these are representative of usual temporal specifications used in software model checking. The software model checkers considered in this work all employ different algorithmic techniques. Further, the features used by MUX are only statistics about structure of programs and properties, and are completely independent of the internal details of the model checkers. These observations give us hope that MUX can be applied successfully to other software model checkers as well. Towards this, we plan to experiment with other software model checkers in future.

The runtimes in this work are all obtained on a single machine. On a different architecture the actual runtimes may vary and the exact gains obtained in this paper too may change. However, the model checking algorithms remain the same and hence similar, if not identical, results are likely to be seen. It is worth noting that MUX being a fully automated technique, it can be easily trained on data obtained from the architecture of interest to the developer.

5. RELATED WORK

Techniques for algorithm selection. SATzilla [47] uses regression to learn an empirical hardness model to predict a SAT solver's performance. It also employs a portfolio-based design [25]. Classification has been used in many works, *e.g.*, [18, 21]. Unlike our work, these approaches use only unweighted classification problems. Thus, instances where there is a significant difference in runtime of two tools are not distinguished from instances where the tools run in almost the same time. Bischi et al. [12] present a generic solution to the algorithm selection problem where they use exploratory landscape analysis to come up with features. Their approach is similar to our approach in that they assign a cost vector to each instance but they use a one-sided support vector regression algorithm. Kadioglu et al. [28] present a SAT solver scheduling approach to boost the performance of the algorithm selection. They propose a nearest neighbour based algorithm to solve the optimization problem arising from the scheduling constraints.

Most of the techniques for algorithm selection, including ours, extract static features from the input instances. Dynamic algorithm selection techniques observe the state of the algorithm at runtime and help it decide which of the possible branches to execute next. Reinforcement learning [31] and classification [40] techniques have been used in dynamic algorithm selection.

Algorithm selection in software engineering. Algorithm selection has seen success in many domains including satisfiability checking [47], quantified-boolean formula solving [40], constraint optimization [41, 12], hardware model checking [14], sorting algorithms [22], compiler optimizations [30, 44], composition [46] and selection [33] of fault localization tools, estimating effectiveness of automated testing tools [16], and selection of domain specific libraries [27] and parallel algorithms [42].

MUX treats the model checkers as black-boxes and derives a many-to-one selector. In order to obtain the best performance from available tools and strategies, white-box approaches aim at fine-tuning software verification tools [11, 4] and some techniques try to derive tool chains of complementary tools [10, 3]. Beyer et al. [11] vary the precision of "merge" and other abstract operations to improve both scalability and precision of model checkers. Apel et al. [4] select abstract domains separately for each variable based on its usage in the program. Conditional model checking [10] proposes sequential composition of model checkers where partial information computed by one checker (which failed to prove/disprove an assertion) is passed to the next. Several tools, including [3], use parallel execution of multiple model checkers.

From a large set of structural features, MUX identified a few features that are statistically correlated with the performance of the model checkers. The usefulness of static code attributes and the relative importance of attributes and data mining algorithms is an active area of research in static defect prediction [35].

6. CONCLUSIONS AND FUTURE WORK

In this work, we presented an algorithmic approach to improve effective use of software model checkers. Our approach synthesizes an algorithm selector by learning from a repository of software verification instances and the performance of the tools on them. We performed a large scale evaluation of our technique on Windows device drivers and showed that the algorithm selector not only reduces the number of timeouts but also reduces the time taken to model check large code bases by a large margin.

Encouraged by the results obtained by MUX for software model checkers for sequential C programs, we want to investigate similar approaches for other software engineering and analysis domains.

We would also like to extend MUX for symbolic and explicit concurrency model checkers.

7. REFERENCES

[1] http://www.cprover.org/boolean-programs.

[2] http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php.

[3] A. Albarghouthi, A. Gurfinkel, Y. Li, S. Chaki, and M. Chechik. UFO: Verification with interpolants and abstract interpretation. In *TACAS*, pages 637–640, 2013.

[4] S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. Rhein. Domain types: Abstract-domain selection based on variable usage. In *HVC*, pages 262–278, 2013.

[5] T. Ball, E. Bounimova, R. Kumar, and V. Levin. Slam2: static driver verification with under 4% false alarms. In *FMCAD*, pages 35–42, 2010.

[6] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *IFM*, pages 1–20, 2004.

[7] T. Ball, V. Levin, and S.K. Rajamani. A decade of software model checking with SLAM. *CACM*, pages 68–76, 2011.

[8] T. Ball and S. K. Rajamani. SLIC: A specification language for interface checking (of C). Technical report, MSR-TR-2001-21, Microsoft Research, 2001.

[9] D. Beyer. Second competition on software verification. In *TACAS*, pages 594–609, 2013.

[10] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking: a technique to pass information between verifiers. In *FSE*, page 57, 2012.

[11] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *CAV*, pages 504–518, 2007.

[12] B. Bischl, O. Mersmann, H. Trautmann, and M. Preuss. Algorithm selection based on exploratory landscape analysis and cost-sensitive learning. In *GECCO*, pages 313–320, 2012.

[13] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*. 2006.

[14] G. Cabodi, S. Nocco, and S. Quer. Thread-based multi-engine model checking for multicore platforms. *ACM Trans. Des. Autom. Electron. Syst.*, 18(3):36:1–36:28, 2013.

[15] C. Chang and C. Lin. LIBSVM: a library for support vector machines. *TIST*, page 27, 2011.

[16] B. Daniel and M. Boshernitsan. Predicting effectiveness of automatic testing tools. In *ASE*, pages 363–366, 2008.

[17] R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin. LIBLINEAR: A library for large linear classification. *J. of ML Research*, pages 1871–1874, 2008.

[18] C. Gebruers, A. Guerri, B. Hnich, and M. Milano. Making choices using structure at the instance level within a case based reasoning framework. In *Int. AI and OR Techn.*, pages 380–386, 2004.

[19] P. Godefroid. Software model checking: The verisoft approach. *FMSD*, pages 77–101, 2005.

[20] Object Management Group. Unified Modeling Language Specification (Version 2.41). 2011.

[21] A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *ECAI*, page 475, 2004.

[22] H. Guo. *Algorithm selection for sorting and probabilistic inference: a machine learning-based approach*. PhD thesis, Kansas State University, 2003.

[23] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, pages 10–18, 2009.

[24] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Model Checking Software*, pages 235–239. 2003.

[25] B. Huberman, R. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, pages 51–54, 1997.

[26] ITU-T. Specification and Description Language (SDL). 2010.

[27] T. Johnson and R. Eigenmann. Context-sensitive domain-independent algorithm composition and selection. In *PLDI*, pages 181–192, 2006.

[28] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm selection and scheduling. In *CP*, pages 454–469, 2011.

[29] M. G. Kendall and J. D. Gibbons. *Rank correlation methods*. Oxford University Press, 1990.

[30] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *PLDI*, pages 171–182, 2004.

[31] M.G. Lagoudakis and M.L. Littman. Algorithm selection using reinforcement learning. In *ICML*, pages 511–518, 2000.

[32] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *CAV*, pages 427–443, 2012.

[33] T. B. Le and D. Lo. Will fault localization work for these failures? An automated approach to predict effectiveness of fault localization tools. In *ICSM*, pages 310–319, 2013.

[34] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Soft. Engg Notes*, pages 1–38, 2006.

[35] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Software Eng.*, 33(1):2–13, 2007.

[36] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: a pragmatic approach to model checking real code. In *OSDI*, pages 75–88, 2002.

[37] A. V. Nori and S. K. Rajamani. An empirical study of optimizations in YOGI. In *ICSE*, pages 355–364, 2010.

[38] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The yogi project: Software property checking via static analysis and testing. In *TACAS*, pages 178–181. 2009.

[39] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[40] H. Samulowitz and R. Memisevic. Learning to solve QBF. In *AAAI*, volume 7, pages 255–260, 2007.

[41] K.A. Smith-Miles. Towards insightful algorithm selection for optimisation using meta-learning concepts. In *IJCNN*, pages 4118–4124, 2008.

[42] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *PPoPP*, pages 277–288, 2005.

[43] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. of the London Math. Soc.*, pages 230–265, 1936.

[44] K. Vaswani, M.J. Thazhuthaveetil, Y.N. Srikant, and P.J. Joseph. Microarchitecture sensitive empirical models for compiler optimizations. In *CGO*, pages 131–143, 2007.

[45] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, pages 203–232, 2003.

[46] S. Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau. Search-based fault localization. In *ASE*, pages 556–559, 2011.

[47] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of AI Research*, pages 565–606, 2008.