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 \textsc{MUX} that addresses the problem of selecting the most suitable software model checker for a given input instance. \textsc{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 \textsc{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, \textsc{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., \cite{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 \cite{20}) and textual (e.g., SDL \cite{26}, JML \cite{33}, SLIC \cite{8}) specification languages. As is true with any promising technology, there are a number of competing tools for software model checking (e.g., \cite{45,19,56,7,38,32,24}). 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 \cite{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 \cite{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 \textit{algorithm selector} \cite{39} for software model checkers. Our technique, called \textsc{MUX}, generates a \textit{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 \textsc{MUX}. The algorithm selector \(\mathcal{AS}\) is synthesized offline by \textsc{MUX} in four steps. The \textit{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. Alternately, the developers who use these tools could also supply them or these can be obtained from benchmarks used in software verification competitions \cite{9}. The model checking problem being undecidable, the model checkers are only semi-algorithms and may not terminate in some cases. Therefore, \textsc{MUX} takes a timeout value \(\Delta\) for running the tools. Software model checkers use complex data structures and
algorithms, and may themselves contain faults, resulting in incorrect output in some cases. MUX therefore also takes the expected results \( r_1, \ldots, r_n \) 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 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 AS synthesized by MUX.

Over the test instances, 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. 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 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 AS selected tools and each of the individual tools on those instances where both finish within the timeout. On these instances, 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, 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
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 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 AS.

The SDV toolkit for Microsoft Windows uses a manually-designed portfolio-based algorithm selector \( \text{AS} \) called Q. Over the test instances, Q could not prevent 51 timeouts, whereas the time taken by 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 \( \varphi_1 \) to \( \varphi_5 \). 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, \( \varphi_1 \) to \( \varphi_5 \), 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 \( \varphi_2 \) and \( \varphi_5 \), 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.

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 in parallel or in a time-sliced manner. 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 in the set of benchmark programs: The optimal choices are underlined. to indicates timeout.

<table>
<thead>
<tr>
<th>Property</th>
<th>SLAM</th>
<th>Yogi</th>
<th>Corral</th>
<th>Q</th>
<th>AVG</th>
<th>AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varphi_1 )</td>
<td>7657s</td>
<td>7192s</td>
<td>4577s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \varphi_2 )</td>
<td>3508s</td>
<td>6737s</td>
<td>3703s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \varphi_3 )</td>
<td>3422s</td>
<td>3647s</td>
<td>237s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \varphi_4 )</td>
<td>3326s</td>
<td>4304s</td>
<td>287s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \varphi_5 )</td>
<td>1649s</td>
<td>3384s</td>
<td>2480s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Any</td>
<td>3912s</td>
<td>4407s</td>
<td>2205s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance</th>
<th>SLAM</th>
<th>Yogi</th>
<th>Corral</th>
<th>Q</th>
<th>AVG</th>
<th>AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (( P_1, \varphi_4 ))</td>
<td>2144s</td>
<td>1580s</td>
<td>1330s</td>
<td>C</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>2 (( P_1, \varphi_5 ))</td>
<td>1650s</td>
<td>1560s</td>
<td>1534s</td>
<td>S</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>3 (( P_2, \varphi_1 ))</td>
<td>1997s</td>
<td>1622s</td>
<td>1622s</td>
<td>S[( \text{to} )]</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>4 (( P_2, \varphi_2 ))</td>
<td>1618s</td>
<td>2801s</td>
<td>3045s</td>
<td>S[( \text{to} )]</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>5 (( P_2, \varphi_4 ))</td>
<td>1967s</td>
<td>1367s</td>
<td>1367s</td>
<td>C[( \text{to} )]</td>
<td>Y[( \text{to} )]</td>
<td></td>
</tr>
<tr>
<td>6 (( P_2, \varphi_5 ))</td>
<td>2416s</td>
<td>2984s</td>
<td>3816s</td>
<td>C[( \text{to} )]</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>7 (( P_2, \varphi_3 ))</td>
<td>2416s</td>
<td>2416s</td>
<td>2416s</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>8 (( P_2, \varphi_6 ))</td>
<td>2077s</td>
<td>2077s</td>
<td>2077s</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>9 (( P_2, \varphi_7 ))</td>
<td>1114s</td>
<td>1114s</td>
<td>1114s</td>
<td>S[( \text{to} )]</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>10 (( P_2, \varphi_8 ))</td>
<td>1897s</td>
<td>3297s</td>
<td>3297s</td>
<td>C[( \text{to} )]</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>11 (( P_2, \varphi_9 ))</td>
<td>2894s</td>
<td>2894s</td>
<td>2894s</td>
<td>S[( \text{to} )]</td>
<td>Y[( \text{to} )]</td>
<td></td>
</tr>
<tr>
<td>12 (( P_2, \varphi_{10} ))</td>
<td>2999s</td>
<td>2999s</td>
<td>2999s</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>13 (( P_2, \varphi_{11} ))</td>
<td>832s</td>
<td>832s</td>
<td>832s</td>
<td>S[( \text{to} )]</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>14 (( P_2, \varphi_{12} ))</td>
<td>1107s</td>
<td>1107s</td>
<td>1107s</td>
<td>C</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>15 (( P_2, \varphi_{13} ))</td>
<td>1049s</td>
<td>1049s</td>
<td>1049s</td>
<td>S[( \text{to} )]</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>16 (( P_2, \varphi_{14} ))</td>
<td>2771s</td>
<td>2771s</td>
<td>2771s</td>
<td>C</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>17 (( P_2, \varphi_{15} ))</td>
<td>1484s</td>
<td>2884s</td>
<td>2884s</td>
<td>C</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

| Timouts | 14 | 8 | 5 | 4 | 10 | 2 |
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 AS synthesized by MUX (see the last column). 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 AS nor Q timeout. On these instances, the tools selected by AS take only 60% 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, 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 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 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 computes aliasing information only during symbolic execution and may scale better on programs with many pointers.

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.

The algorithm selection problem can be formulated as a multi-class classification problem as follows: Given a program-property pair \((P, \varphi)\) 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-
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 (δ).

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 \( (P, \varphi) \), 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 (δ). The discretized variant of it is denoted as “with weights”. The discretized variant 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 (i.e., outputs of MUX) are an algorithm selector \( AS \).

<table>
<thead>
<tr>
<th>Machine learning algorithm</th>
<th>ML problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-linear SVM without weights</td>
<td>3 binary, 1 multi-class</td>
</tr>
<tr>
<td>Non-linear SVM with weights</td>
<td>3 binary, 1 multi-class</td>
</tr>
<tr>
<td>Non-linear SVM with discretized weights</td>
<td>3 binary, 1 multi-class</td>
</tr>
<tr>
<td>Linear SVM without weights</td>
<td>3 binary, 1 multi-class</td>
</tr>
<tr>
<td>Linear SVM with weights</td>
<td>3 binary, 1 multi-class</td>
</tr>
<tr>
<td>Linear SVM with discretized weights</td>
<td>3 binary, 1 multi-class</td>
</tr>
<tr>
<td>Linear regression</td>
<td>Ridge regression</td>
</tr>
</tbody>
</table>

Total number of problems: 25

### 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 \( AS \).

1. \( DB \leftarrow \emptyset \) // Initialize the repository
2. \( DB \leftarrow DB \cup \{ (P_1, \varphi_1) \} \)
3. for each program-property pair \( (P_i, \varphi_i) \) do
   4. for each model checker \( T_j \) do
      5. \( Time_{ij} \leftarrow Run_{ij} \) to time \( \Delta \) to check \( P_i \models \varphi_i \)
      6. if \( Time_{ij} < \Delta \) then
         7. \( Result_{ij} \leftarrow \text{the result of } T_j \text{ on } (P_i, \varphi_i) \)
      8. if \( Result_{ij} \neq t \) then \( Time_{ij} \leftarrow 2 \times \Delta \)
   9. \( end \)
10. \( FV_i \leftarrow \text{Extract structural features from } (P_i, \varphi_i) \)
11. \( DB \leftarrow DB \cup \{ (FV_i, Time_{ij}, \ldots, Time_{im}) \} \)
12. \( end \)
13. \( TS \leftarrow \text{Select a subset of } \{ (P_1, \varphi_1), \ldots, (P_n, \varphi_n) \} \)
14. \( VS \leftarrow \{ (P_1, \varphi_1), \ldots, (P_n, \varphi_n) \} \setminus TS \)
15. \( F \leftarrow \text{Select a subset of most useful features using } TS \)
16. Train classification models on \( DB \) restricted to \( TS \) and \( F \)
17. Train regression models on \( DB \) restricted to \( TS \) and \( F \)
18. Select the best ML model
19. \( C \leftarrow \text{Select the best classification model wrt } VS \)
20. \( R \leftarrow \text{Select the best regression model wrt } VS \)
21. \( AS \leftarrow \text{Synthesize } B \text{ as a program from feature vectors to identifiers of the model checking tools} \)

22. \( end \)
on \((P_i, \varphi_i)\). If \(T_j\) does not terminate before the timeout occurs then \(M_{UX}\) aborts the model checking run, setting \(Time_{\ell j}\) to the timeout value \(\Delta\). If \(T_j\) finishes before the timeout (line 9) but its result is incorrect (line 8) then \(M_{UX}\) resets \(Time_{\ell j}\) to a value higher than \(\Delta\). In this work, we set it to \(2 \times \Delta\) but this is configurable. \(M_{UX}\) stores the feature vector \(FV_i\) for \((P_i, \varphi_i)\) 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, we consider a program-property pair \((P_i, \varphi_1)\) belonging to the validation set such that some \((P_i, \varphi_j)\) is present in the training set. We note that the set of features pre-dominantly include features about the strength of linear correlation between two statistical variables. 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 \(M_{UX}\) during the offline phase:

(RQ1) Selection of important features – Which features are identified by \(M_{UX}\) 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 \(M_{UX}\) as the algorithm selector \(AS\)?

(RQ3) Performance and scalability – How does \(M_{UX}\) perform in the training and validation steps?

Next, we evaluate effectiveness of the algorithm selector \(AS\) synthesized by \(M_{UX}\) in the online phase:

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

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, \(M_{UX}\) generates one encoding each for every tool. The encoding of an instance \((FV_i, Time_{e1}, \ldots, Time_{eim})\) for a tool \(T_j\) is simply \((FV_i, Time_{\ell j})\) 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.

\(M_{UX}\) 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, \(M_{UX}\) assigns the score for a prediction as runtime of the predicted tool. Given the feature vector \(FV_i\) of an instance \(i = (FV_i, Time_{e1}, \ldots, Time_{eim})\), if a classification model selects label \(\ell\) then the score is \(S(i, \ell) = Time_{\ell j}\). The aggregate score of a classification model \(C\) over the validation set is equal to \(\sum_{i \in VS} S(i, C(FV_i))\). 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 \(C_1\) with 90% precision may take more time to finish the tasks than those predicted by another classification model \(C_2\) with, say only 80% precision. In such a case, \(M_{UX}\) prefers \(C_2\) over \(C_1\).

As discussed in Section 3.3, \(M_{UX}\) trains one regression model for each of the tools. The prediction of the combined regression model \(R\) on an instance \(i\) is the tool with smallest predicted runtime. Suppose \(\ell\) is the tool predicted by \(R\). The definitions of score \(S(i, \ell)\) and aggregate score here are the same as the definitions in the classification case. \(M_{UX}\) then selects the best model, by aggregate score, between the best classification model and the best regression model (line 20). Finally, the algorithm selector \(AS\) 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 \(M_{UX}\). 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].
Table 5: Features identified by MUX as important for algorithm selection and their correlation coefficients

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

(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 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 representative 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 4.2.

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.

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 function 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 5, 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 a 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, \( AS \).

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), \( AS \) predicts one of them algorithmically. Besides, as our results on the test instances would show, \( AS \) 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 modeled 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 \( AS \) 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 \( AS \) 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 \( i \)th row and the tool labeling the \( j \)th column timeout. A diagonal entry \((i, i)\) gives the total timeouts for the \( i \)th tool. The matrix is symmetric and hence only values in the lower triangle are given. The column labeled “optimal” corresponds to the optimal timeout. As can be seen, the time taken by \( AS \)’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, \( AS \) 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 \( AS \)’s choices. We highlight that \( AS \) does not degenerate into selecting Corral by default. \( AS \) selects Corral only on 44% instances in the test data. Further, as discussed earlier, the number of timeouts that Corral results in is much higher compared to \( AS \). Q reduces the number of timeouts compared to any individual model checker (though still higher than \( AS \)). Unfortunately, it does not show any improvement with respect to the total runtime. As seen from Table 7, \( AS \)’s choices take only 78% time compared to Q’s choices. At this point, the performance of \( AS \) is somewhat farther from the optimal time. It is 31% slower than the algorithm selector which always selects the optimal tool.

(RQ5) Productivity gain

We want to now compare the total time taken by the tools selected by \( AS \) 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 \( AS \)’s choices both finish within the timeout. As can be seen, the time taken by \( AS \)’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, \( AS \) 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 \( AS \)’s choices. We highlight that \( AS \) does not degenerate into selecting Corral by default. \( AS \) selects Corral only on 44% instances in the test data. Further, as discussed earlier, the number of timeouts that Corral results in is much higher compared to \( AS \). Q reduces the number of timeouts compared to any individual model checker (though still higher than \( AS \)). Unfortunately, it does not show any improvement with respect to the total runtime. As seen from Table 8, \( AS \)’s choices take only 78% time compared to Q’s choices. At this point, the performance of \( AS \) 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 \( AS \) in the online phase is very small. On the test data, \( AS \) typically takes a few sec-

Table 7: Pairwise comparison of timeouts

<table>
<thead>
<tr>
<th>Tool</th>
<th>Instances</th>
<th>Time</th>
<th>AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>2883</td>
<td>2d 09h</td>
<td>3d 03h (131%)</td>
</tr>
<tr>
<td>Q</td>
<td>2859</td>
<td>3d 05h</td>
<td>2d 12h (78%)</td>
</tr>
<tr>
<td>Corral</td>
<td>2844</td>
<td>2d 19h</td>
<td>2d 15h (94%)</td>
</tr>
<tr>
<td>Yogi</td>
<td>2839</td>
<td>4d 04h</td>
<td>2d 18h (68%)</td>
</tr>
<tr>
<td>SLAM</td>
<td>2717</td>
<td>3d 06h</td>
<td>1d 19h (55%)</td>
</tr>
</tbody>
</table>

Table 8: Comparison of the total runtime

<table>
<thead>
<tr>
<th>Tool</th>
<th>Instances</th>
<th>Time</th>
<th>AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>2883</td>
<td>2d 09h</td>
<td>3d 03h (131%)</td>
</tr>
<tr>
<td>Q</td>
<td>2859</td>
<td>3d 05h</td>
<td>2d 12h (78%)</td>
</tr>
<tr>
<td>Corral</td>
<td>2844</td>
<td>2d 19h</td>
<td>2d 15h (94%)</td>
</tr>
<tr>
<td>Yogi</td>
<td>2839</td>
<td>4d 04h</td>
<td>2d 18h (68%)</td>
</tr>
<tr>
<td>SLAM</td>
<td>2717</td>
<td>3d 06h</td>
<td>1d 19h (55%)</td>
</tr>
</tbody>
</table>

(j = 2), SLAM times out on 166 instances on which the tools selected by \( AS \) do not timeout, whereas the tools selected by \( AS \) timeout on 9 instances on which SLAM does not timeout. Thus, with respect to SLAM, \( AS \) 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 \( AS \) results in timeout but Q does not and 24 instances where Q results in timeout but \( AS \) does not. Overall, \( AS \) successfully eliminates a significant number of timeouts originating from each of the model checkers and the choices of Q.
onds 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. AS. 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 AS caused 44 timeouts, better than individual tools and Q. For the non-timeout cases, 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 instrumentation bugs in the tools. This was based on the explicit tagging by the developers who wrote the device drivers. The tool designers have confirmed this. 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 validity 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 [12] uses regression to learn an empirical hardness model to predict a SAT solver’s performance. It also employs a portfolio-based design [23]. 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. Kadiglu et al. [23] 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 [33] 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 [30], constraint optimization [41] [12], hardware model checking [14], sorting algorithms [22], compiler optimizations [30] [44], composition [46] and selection [53] 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] [5]. 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 [19] 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 [45].

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