2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)

Learning to Optimize the Alloy Analyzer

Wenxi Wang*, Kaiyuan Wang†, Mengshi Zhang* and Sarfraz Khurshid*
*University of Texas at Austin
{wenxiw,mengshi.zhang,khurshid}@utexas.edu
†Google Inc.
kaiyuanw@google.com

***Abstract*—Constraint-solving is an expensive phase for scenario finding tools. It has been widely observed that there is no single "dominant" SAT solver that always wins in every case; instead, the performance of different solvers varies by cases. Some SAT solvers perform particularly well for certain tasks while other solvers perform well for other tasks. In this paper, we propose an approach that uses machine learning techniques to automatically select a SAT solver for one of the widely used scenario finding tools, i.e. Alloy Analyzer, based on the features extracted from a given model. The goal is to choose the best SAT solver for a given model to minimize the expensive constraint solving time. We extract features from three different levels, i.e. the Alloy source code level, the Kodkod formula level and the boolean formula level. The experimental results show that our portfolio approach outperforms the best SAT solver by 30% as well as the baseline approach by 128% where users randomly select a solver for any given model.**

***Index Terms*—Alloy Analyzer, SAT solver, machine learning**

I. Introduction

Writing declarative models and specifications has numerous benefits, ranging from automated reasoning and correction of design-level properties before systems are built [1], to automated testing and debugging of the implementations after systems are built [2]. Alloy [3] is one of the well-known scenario finding tools that model system properties. Alloy models are declarative and expressive enough to capture the intricacies of real systems. Alloy comes with an analyzer which provides an automatic analysis engine based on off-the-shelf SAT solvers [4] and it is able to generate valuations for the relations in the models such that the properties modeled hold or are refuted as desired. The powerful Alloy analysis has motivated its use in a wide range of applications, including security [5], networking [6] and UML analysis [7].

Alloy supports first-order relational logic with transitive closure. The Alloy analyzer is able to analyze Alloy models which consist of relational expressions/formulas under user-defined scopes. Internally, the analyzer translates the Alloy model into Kodkod formulas [8], which in turn is translated into the boolean formulas. Finally, the boolean formulas are fed into a SAT solver to find a solution which is then mapped back to an Alloy instance for analysis. In this paper, we refer the Alloy source code level as level 1, the Kodkod formula level as level 2, and the boolean formula level as level 3.

Typically, the SAT solving time takes a majority of the analysis time and is often the bottleneck for the end-to-end time. As the scope of the model becomes larger, Alloy's analyzing ability drops dramatically because of the expensive SAT solving. We observed that there is no single "dominant" SAT solver that always win in every model. Instead, the performance of different solvers varies by models. This paper aims to alleviate the expensive SAT solving by helping the users to pick the SAT solver that achieves the best performance given an arbitrary Alloy model. The idea is to extract features from the model and use a machine learning model to predict which SAT solver is more likely to solve the problem in the minimum amount of time from a set of component solvers.

Our technique has four phases: (1) feature extraction phase; (2) feature selection phase; (3) training phase; and (4) testing phase. In the feature extraction phase, we extract features from all 3 levels of a given Alloy model, including the Alloy source code level, the Kodkod formula level and the boolean formula level. These features are all static and fast to extract. We extract the number of different operators at the source code level (e.g. set union), the Kodkod formula level (e.g. n-nary expression and relational bounds) and the boolean formula level (e.g. not gate). Additionally, we also collect the metrics of an AST, e.g. the height, diameter and total number of nodes, across all 3 levels. The feature extraction phase is applied before the training and testing phase. We only focus on static features to avoid the overhead of extracting the dynamic features from invoking the SAT solver. In the feature selection phase, we evaluate the importance of the features in each level and only select the ones that make good impacts. In the training phase, we extract features of various models with different scopes. These models are run against multiple SAT solvers we collected from the SAT competition [9] and all running times are collected to label different models with various scopes. Then, we apply Adaptive Boosting (AdaBoost) learning model to learn the best performance SAT solver for each model and scope. In the testing phase, we extract features of unseen models with different scopes and use the learned model to predict the best SAT solver and compare the result against each component solver, the random solver selection, and the best solver selection.

The experimental results show that our technique outperforms the baseline approaches significantly, including 30% acceleration of the best on average component solver, 2.28 times the speed of random solver selection and 0.62 times the speed of the best solver selection.

This paper makes the following contributions:

- the first (as far as we know) portfolio approach proposed

978-1-7281-1736-2/19/$31.00 ©2019 IEEE
DOI 10.1109/ICST.2019.00031

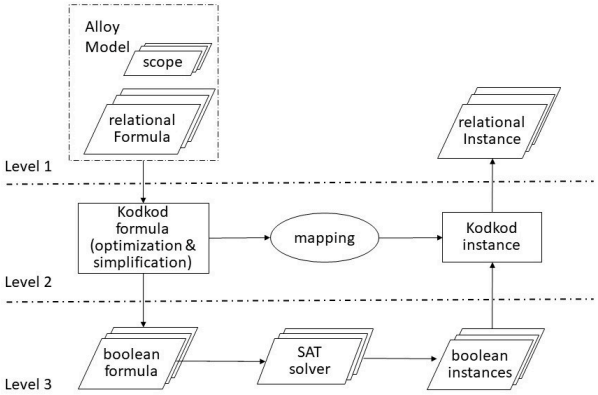

Fig. 1: Architecture of the Alloy analyzer version 4.2.

for the Alloy Analyzer.

- feature extraction from three levels in the Alloy Analyzer; and evaluated the feature importance in each level.
- results suggesting that, our machine learning based portfolio approach is promising and can be a good direction for further development.

II. Background

A. Alloy Analyzer

Alloy is a declarative language for lightweight modeling and software analysis. The language is based on first-order logic with transitive closure. Alloy comes with an analyzer which is able to perform a bounded exhaustive analysis under a given scope. The input of the Alloy analyzer is an Alloy model that describes the system properties. Users write Alloy models with relational expressions and formulas, and provide commands with given scopes bounded on the universe of discourse. The analyzer translates the commands with scopes into conjunctive normal form (CNF) and invokes an off-the-shelf SAT solver to search for solutions, i.e. boolean instances. The boolean instances are then mapped back to Alloy level instances that contain valuations of all relations and displayed to the end users. The overall architecture of Alloy Analyzer is shown in Figure 1. The architecture can be divided into three levels: the Alloy source code level (level 1), the Kodkod formula level (level 2), and the boolean formula level (level 3). As of version 4.2, the Alloy analyzer invokes the Kodkod model-finder to translate formulas between levels 2 and 3, where users write models in the analyzer at level 1.

1) Alloy Models: An Alloy model consists of a set of relations (e.g. signatures, fields and variables) and constraints (e.g. predicates, facts and assertions). A signature defines a set of atoms. A field maps each atom of a signature to other atoms. Note that a signature is a special relation of arity 1 and a field is a relation of arity > 1. Users can also introduce new relations using variable declarations, e.g. parameters or let expressions. A predicate defines a boolean formula that evaluates to either true or false. Users can use the Alloy's built-in `run` command to invoke a predicate and the Alloy

```
1. sig List {
2.   // header: List->Node is a partial function
3.   header: lone Node }
4. sig Node {
5.   // link: Node->Node
6.   link: one Node }
7. pred Acyclic1(l: List) {
8.   all n: l.header.*link | n !in n.^link }
9. pred Acyclic2(l: List) {
10.  no l.header || some n: l.header.*link | no n.link }
11. check { // command
12.  all l: List | Acyclic1[l] <=> Acyclic2[l]
13. } for 6
```

Fig. 2: An example Alloy model that checks if two different ways of modeling acyclicity constraint on a list are equivalent.

analyzer either returns an instance if the predicate is satisfiable or reports that the predicate is unsatisfiable. A fact is a boolean formula that is implicitly enforced to be true by the Alloy analyzer. An assertion is a boolean formula that is used with the built-in `check` command to check if any counter example can refute the asserted formula. If the assertion is valid, then no counter-example is found; otherwise, an counter-example is reported. All analysis are performed in a bounded scope.

Figure 2 shows an Alloy model which checks if two acyclicity predicate of a list (`Acyclic1` in lines 7-8 and `Acyclic2` in lines 9-10) are semantically equivalent. The signature `List` (lines 1-3) declares a set of list atoms. The signature `Node` (lines 4-6) declares a set of node atoms. The `header` field (line 3) is a partial function and maps each list atom to at most one node atom. The `link` field (line 6) is a function and maps each node atom to exactly one node atom. `Acyclic1` states that for every node `n` which is reachable from the parameter list `l`'s header following zero or more traversals along the `link`, `n` is not reachable from itself following one or more traversals along the `link`. `Acyclic2` states that either the parameter list `l` has no header or there exists some node `n` reachable from `l`'s header following zero or more traversals along the `link` such that `n` does not have a subsequent node along the `link`. The `check` command (lines 11-13) checks if `Acyclic1` and `Acyclic2` are semantically equivalent for every list `l` up to 6 lists and 6 nodes.

2) Kodkod Translation: Kodkod [8] is an efficient SAT-based model finder which is able to specify partial solutions, i.e., a priori partial but exact knowledge about a problem's solution. It is able to effectively detect and break symmetric formulas. Internally, Kodkod translates Alloy code into Kodkod formulas, which is then translated to boolean formulas and fed into a SAT solver. Kodkod achieves symmetry breaking by a symmetry detection algorithm that works in the presence of partial solutions, a sparse-matrix representation of relations, and a compact representation of boolean formulas. These techniques make Kodkod effective in translating relational logic into boolean formulas and alleviating the burden of existing SAT solvers by removing redundant constraints.

Note that Kodkod converts the scope of Alloy commands with the notion of relational bounds. A bounded relational specification is a collection of constraints on relational vari-

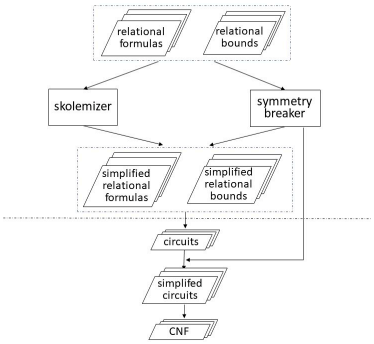

Fig. 3: Translation from relational logic to CNF in Kodkod

ables that are bounded by relational constants (i.e., sets of tuples). All bounded constants consist of tuples that are drawn from the same finite universe of discourse. The upper bound specifies the tuples that a relation may contain; the lower bound specifies the tuples that the relation must contain. Thus, the scope of the command in an Alloy model is translated into the relational bounds with lower bound of 0 and upper bound of the given scope. For the example in Figure 2, the scope 6 is translated such that Kodkod will only consider the search space of each signature from 0 to 6 atoms.

The translation from Kodkod formulas to boolean formulas refers to level 2 and level 3, respectively in Figure 1. The translation process inside Kodkod is depicted in Figure 3. The Kodkod formula and bounds are firstly simplified and optimized by symmetry breaking and skolemizing, which tighten the bounds and eliminate the top-level predicates. The optimized relational formula and bounds are then translated into a circuit, which are further augmented with a symmetry breaking predicate that eliminates any remaining symmetries. Finally the optimized circuit is translated to CNF formula. Note that the part above the horizontal dash line in Figure 3 corresponds to level 2 in the Alloy analyzer, while the part below the dash line corresponds to level 3.

3) Back-end SAT Solvers: After the translation to boolean formulas in Kodkod, off-the-shelf SAT solvers can be used to solve the boolean formulas. The input of the SAT solver is a formula ψ in CNF, that is, a conjunction of clauses. Each clause c consists of a disjunction of literals. A literal ℓ is either a variable b, or its complement $\neg b$. For instance in Example II.1, CNF formula ψ_1 includes two clauses c_1 and c_2 where four variables b_1, b_2, b_3 and b_4 are included.

Example II.1. $\psi_1 = c_1 \wedge c_2$; $c_1 = b_1 \vee \neg b_3$; $c_2 = b_2 \vee b_4 \vee b_3$;

In the last decades, the performance of SAT solvers has increased dramatically with the invention of advanced searching and learning strategy, and the data structures that allow efficient implementation of search space pruning. However, the SAT solving ability is still the core challenge and bottleneck for applications like the Alloy analyzer. Therefore, reducing the SAT solving time is an important aspect in improving the application user experience. To achieve this, one way is to increase the solving efficacy of a particular SAT solver. Another way is to integrate the solving strength of different solvers to introduce a "solver" which is more powerful than any of the component solvers.

The International SAT Solver Competition [9] is an established series of competitive events aiming at objectively evaluating the progress in state-of-the-art procedures for solving boolean satisfiability (SAT) instances. Over the years, the competitions have significantly contributed to the fast progress in SAT solver technology. Our idea is to combine the state-of-the-art SAT solvers in the competition to propose a more powerful portfolio SAT solver for the Alloy analyzer.

B. Portfolio Solvers with Machine Learning

The essential problem behind the portfolio solver development is the algorithm selection problem. The "No Free Lunch" (NFL) theorems [10] state that no algorithm can be the best across all possible problems and that on average all algorithms perform the same. This makes the algorithm selection essential for improving the problem solving capability. The idea is to select different algorithms best for different parts of the problem space.

In the area of combinatorial search problems, such as constraint satisfaction problems (CSP), satisfiability (SAT) and satisfiability modulo theories (SMT) problems, solvers indeed comply with the NFL theorems and many portfolio solvers are proposed in response to that trying to gain the optimal world. There is a trade-off between risk (the variability in solving performance) and reward (the expected solving performance) that we need to make a choice. An efficient portfolio is the one that has the highest possible reward for a given level of risk, or the lowest risk for a given reward [11].

The searching in solvers is usually very complicated and unpredictable in different cases even to the authors who invent those solvers. Given this, finding a way to generalize the searching and solving behaviors of the solvers in different problem cases with the big statistical data they produce during their solving history seems promising. In addition, classifying the component solvers based on their outperformed solving history in certain input problem instances to make them work only in their specialties is very demanding in portfolio solvers. Nevertheless, this is under the assumption that the solvers would behave similarly in similar recognized problem cases. Given the above two points, machine learning approach, which uses statistical techniques to give systems the ability to "learn" with data, without being explicitly programmed, are highly suitable for developing portfolio solvers.

III. Motivation

Table I shows the run time result (time in seconds) among top 5 solvers in the International SAT Competitions 2018, namely MapleLCMDistChronoBT (MapleL1), Maple_LCM_Scavel_fix2 (MapleL2), Maple_CM (MapleC1), cms55-main-all4fixed (cms55), and Maple_CM_ordUIP (MapleC2) in order of ranking in the competition. The Alloy

TABLE I: Solving time in seconds for sample Alloy models with different SAT solvers.

Model	cms55	MapleC1	MapleC2	MapleL1	MapleL2	Optimal
abstMem	49.7	**37.6**	47.3	80.7	300	37.6
addrsB1h	45.2	14.8	17.1	**12.3**	32.4	12.3
filesystem	69.7	88.7	87.7	46.2	**25.0**	25.0
grandpa1	**3.3**	14.6	17.3	14.2	26.9	3.3
hotel1	40.1	**10.7**	12.4	94.8	62.1	10.7
lists	78.1	**50.5**	59.1	124.9	91.8	50.5
p300hot	**27.2**	104.7	103.7	149.5	300	27.2
sum	313.3	321.7	344.6	522.4	838.1	166.6

models we use are from the sample models in the Alloy 4.2 distribution. `abstMem` models the abstract memory and we use the check command `check$2` with scope 30 for solving. `addrsB1h` models the address book and we use the check command `delUndoesAdd` with scope 11. `filesystem` models a file system and we use the check command `check$1` with scope 11. `grandpa1` models the grandpa puzzle and we use the check command `NoSelfGrandpa` with scope 42. `hotel1` models guest accessing rooms in a hotel and we use the check command `NoBadEntry` with scope 9. `p300hot` models possible intruders in a hotel and we use the check command `NoIntruder` with scope 13.

We have the following three observations based on the results in Table I: 1) there is no best solver in all cases; and each solver could be the best for certain cases. 2) the solving time differs quite a lot among the solvers, although the solving behaviors of MapleC1 and MapleC2 are similar. The ideal optimal portfolio solver would take 166.6 seconds, which outperforms each component solver dramatically. 3) the solver that wins in a majority of times in the competition does not always perform the best for a majority of our Alloy model samples. This result is consistent with our further experimental results in Section V. All of the above observations makes selecting the best SAT solver for different Alloy models even more important. Moreover, the portfolio approach would reduce the end to end time of Alloy users by selecting the fastest solvers for models with larger scope, which helps the users to gain more confidence in the checked model properties.

IV. Back-End SAT Solver Selection

We take the solver selection in our portfolio approach as the classification problem using supervised machine learning strategy. We aim to classify each component solver based on the problem cases in which it wins. To make fully use of the machine learning techniques, we need to feed it with enough sample cases whose features are extracted from corresponding Alloy models. Additionally, we use the CNF files translated from these Alloy models by Kodkod as the input of different SAT solvers and collect the SAT solving time. The outperforming solvers serve as the predicted labels for the machine learning classifiers. Finally, we use the collected data to train machine learning models and try to avoid the over-fitting issues. The machine learning models can be then used to predict the best solvers for unseen models.

A. Case Generation

As mentioned in Section II, the scope in each command sets the bound of each signature in Alloy model. Therefore, as the scope in the Alloy models increases, the complexity of the problem grows exponentially which becomes a challenge even for the most advanced SAT solvers. In order to obtain more data and models with different complexities, we not only collect all different sample models from Alloy distribution and existing works [12], [13], but also use various scopes for the same model. For each model, we increase the scope of one signature by one at a time, until the scopes of all the signatures reach to the maximum scope we set up. In this case, for one Alloy model with c commands and f signatures, with the initial overall scope for all signatures is, and maximum scope ms, there are $c*(ms-is+1)^f$ potential sample cases. Note that the cases here refers to the CNF input files generated from different Alloy models with different scopes. Each case is indexed with $\mathfrak{m}$_$\mathfrak{c}$_$\mathfrak{s}$ where $\mathfrak{m}$ refers to the Alloy model name, $\mathfrak{c}$ refers to the command name and $\mathfrak{s}$ refers to the scope.

B. Feature Extraction

We extracted the features from all three levels in Alloy as shown in Figure 1, aiming to get the machine learning model a complete picture of how the input Alloy models look like in three perspectives. Note that we only collect the static features which cost little overhead for our portfolio solver. We also evaluate the feature extraction time, as well as how the features in each level contribute to the portfolio approach in Section V. Details about how we extract the features in each level are illustrated in the following Sections IV-B1, IV-B2 and IV-B3. In addition, we discuss the feature refinement in Section IV-B4.

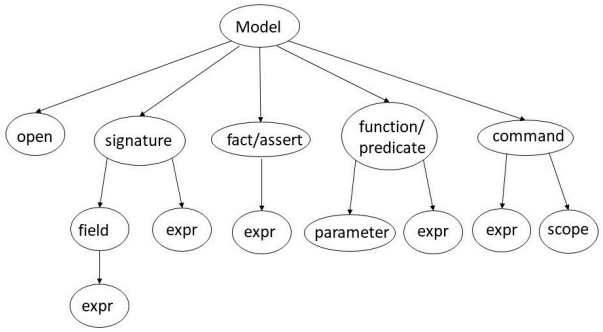

Fig. 4: The General AST Node Classification in Alloy

1) Feature Extraction in Level 1: We collect the features in level 1 (denoted as $\mathfrak{X}$) from the Abstract Syntax Tree (AST) at the Alloy source code level to capture the model characteristics at a higher level.

Figure 4 shows the general AST structure of an Alloy model. Note that we distinguish each node with each other even though they are in the same kind. For instance, the "expr" node under the "fact/assert" node is different from the

"expr" node under "function/predicate" node. The reason is that we want to collect more detailed information which may help the machine learning techniques learn the internal rules. Table II shows specifically how the expressions ("expr" node in Figure 4) look like. We further differentiate the expressions (Expr) from the ones that return a boolean value which we call formulas (Form). For detailed Alloy grammars, please refer to the Alloy tutorials [14].

We count the occurrence number of each node in Figure 4 and the occurrence of each kind of operators in normal expressions and formulas respectively. In addition, the height, the diameter, and the total node number of the AST, together with the scope generated in Section IV-A are also extracted as features in this level. There are in total 123 features. Note that different commands in the same Alloy model invoke different parts of the AST. Thus, the extracted features are also different for different commands.

2) Feature Extraction in Level 2: We collect the features in level 2 (denoted as $\mathfrak{Y}$) from the AST of the simplified Kodkod formulas, and the simplified relational bounds in Figure 3. The goal is to capture the model complexity information in the level just before the Kodkod formulas are translated into boolean formulas. The feature kinds extracted from the Kodkod formulas are similar to the ones in level 1 but with different operator types. Additionally, AST nodes at the Kodkod formula level contain relational bound which are different from the ones in Alloy source code level.

We extract features from the Kodkod ASTs based on the node types. More details of the Kodkod AST node types can be found in the the Kodkod documentation [15]. We choose 42 features in total which fall into four categories:

- The total number of expressions, the total number of integer expressions, the total number of formulas; the total number of the AST nodes; the height of the AST; the diameter of the AST.
- The number of constants, variables and declarations.
- The ratio of the integer expressions and expressions, and the ratio of the quantified formulas and any formulas; the ratio of the expressions and formulas; the ratio of the relations and predicates.
- The number of each specific kind of expressions, formulas and declarations.

The relational bound in Kodkod defines the bound on each relation. The number of each relation in different Alloy model is different. In addition, there is no natural order in the relational bounds. In order to extract features for machine learning models, we need to properly convert the relational bounds into a fixed length feature vector. We sort the relational bounds in descending order, with the intuition that larger bound values indicate a higher complexity of the models. Therefore, keeping larger bound values is better than keeping smaller bound values. Then, we make the feature vectors fixed length ℓ by doing the following steps. For models that have shorter feature vectors (derived from the bound values), we append zeros to the end of the feature vectors. This strategy can be interpreted in a way that the bound of each non-existing signature is zero. For models that have longer feature vectors, we use the top ℓ bound features and truncate the remaining features. Formally, we introduce a sorting function $\mathcal{S}([b_{i_1}, b_{i_2}, .., b_{i_B}])$ as

$$\mathcal{S}([b_{i_1}, b_{i_2}, .., b_{i_B}]) = [b_{j_1}, b_{j_2}, .., b_{j_B}],$$

where $[b_{j_1}, b_{j_2}, .., b_{j_B}]$ is a list of relational bounds such that

$$\forall j_x, j_y \in \{j_1, j_2, .., j_B\}(x \leq y \iff b_{j_x} \geq b_{j_y})$$

The relational bound feature vector extraction function is defined as

$$\mathcal{J}_w(\mathcal{S}([b_{i_1}, b_{i_2}, .., b_{i_B}])) = \begin{cases} [b_{j_1}, b_{j_2}, .., b_{j_\ell}], & |B| \geq \ell \\ [b_{j_1}, b_{j_2}, .., b_{j_B}] \circ \mathbf{0}_{\ell - B}, & |B| < \ell \end{cases}$$

where $\circ\ \mathbf{0}_{\ell-B}$ means padding $\ell - B$ zeros after the corresponding feature vectors to length ℓ.

Besides the feature vectors derived from the bound values, we also take 1) the total bound value of all the relation nodes in AST; 2) and the ratio of the total bound and the relations, as two additional bound features.

3) Feature Extraction in Level 3: The features in level 3 (denoted as $\mathfrak{Z}$) are extracted from the CNF formula in Figure 3, which includes the total number of boolean variables and clauses respectively, and the ratio of the variable number and clause number. Note that, the reason we do not extract features from the simplified circuits after the further symmetry breaking (refer to Figure 3) is that traversing the heavy circuits is exponentially time-consuming, even more expensive than the whole translation time from level 1 to level 3.

4) Feature Refinement: Some features we collected from all three levels are the same across all models and thus we assume these features would have less opportunity to provide useful information for the machine learning classifiers to differentiate models with different complexities. We remove these features from our feature list.

Furthermore, we investigate how the features in each level contribute to the machine learning models based on ablation studies. We select the features which make positive contribution to the prediction results. The experimental results are shown in Section V.

TABLE II: Expressions in Alloy Abstract Syntax Tree

Expression		Descriptions
Call Expr/Form		predicate/function invocation
quant	Expr	sum, comprehension
	Form	all, no, some, one, lone
list	Expr	disjoint, total order
	Form	and, or
binary	Expr	→, . , <: , :>, ++, set operators(+, -, &), arithmetic operators (plus, minus, *, /, remainder, ≪, ≫, ⋙)
	Form	=, !=, implies, < , <=, >, >=, in, !in, and, or, iff
unary	Expr	set, lone, one, some, exactly, ∼, ^ , *(transitive closure)
	Form	lone, one, no, some, not
ITE Expr/Form		implies else

TABLE III: Format of Raw Data.

$\mathfrak{m\_c\_s}$	$\mathfrak{X}_1$	...	$\mathfrak{X}_{xn}$	$\mathfrak{Y}_1$	...	$\mathfrak{Y}_{yn}$	$\mathfrak{Z}_1$	...	$\mathfrak{Z}_{zn}$	$\mathfrak{S}_1$	...	$\mathfrak{S}_{sn}$	Label
t_1	e_1^1	...	e_{xn}^1	f_1^1	...	f_{yn}^1	g_1^1	...	g_{zn}^1	st_1^1	...	st_{sn}^1	s_1
t_2	e_1^2	...	e_{xn}^2	f_1^2	...	f_{yn}^2	g_1^2	...	g_{zn}^2	st_1^2	...	st_{sn}^2	s_2
t_3	e_1^3	...	e_{xn}^3	f_1^3	...	f_{yn}^3	g_1^3	...	g_{zn}^3	st_1^3	...	st_{sn}^3	s_1
t_4	e_1^4	...	e_{xn}^4	f_1^4	...	f_{yn}^4	g_1^4	...	g_{zn}^4	st_1^4	...	st_{sn}^4	s_3
					...								

C. Data for Learning

Table III gives the format of our raw data. Each data sample corresponds to a case ($\mathfrak{m\_c\_s}$) which includes the values for each feature type $\mathfrak{X}$, $\mathfrak{Y}$, and $\mathfrak{Z}$, the solving time of each component solver ($\mathfrak{S}$) and the solver that wins for that case according to the solving time (Label). The format of the input training data for machine learning is the raw data sample excluding the solving time columns ($\mathfrak{S}$).

In this paper, the solving time is the most important target we want to optimize. However, sometimes the solving time for multiple solvers varies only in a couple of seconds. Moreover, the small solving time difference might be due to many factors such as I/O and machine workload. To mitigate this issue, we setup a solving time threshold to determine if multiple solvers perform similarly in each sample case. The threshold is defined as follows. We first find the minimum solving time among all component solvers for each sample case (indexed by i), which is denoted by $m_{st} = min(st_1^i, ...st_{sn}^i)$. Then, we choose a percentage p and compute a so called "relative" solving time $\mathfrak{rt}(p) = m_{st} * (1 + p)$ which we treat as one maximum solving time under which multiple solvers perform equally well. Next, we choose an absolute value v and compute a so called "absolute" solving time $\mathfrak{at}(v)$ which we treat as another maximum solving time under which multiple solvers perform equally well. Finally, we compute the threshold as $\mathfrak{t} = max(\mathfrak{rt}, \mathfrak{at})$. Note that all the solving time is in seconds.

We use the threshold $\mathfrak{ft}$ to filter out the samples where the solving time of *all* SAT solvers is below $\mathfrak{ft}$. For example, suppose we have a sample that includes the solving time vector [1.8, 0.9, 1.2] of three component solvers, with $p = 10$ and $v = 1$. The minimum solving time is 0.9, thus $\mathfrak{rt}(p) = 0.99$, $\mathfrak{at}(v) = 1.9$, and $\mathfrak{ft} = 1.9$. Since all the three solving times (1.8, 0.9 and 1.2) are less than $\mathfrak{ft}$, this sample will be eliminated from our dataset.

Additionally, we use the threshold to decide fairly if the machine learning model selects the right solver. Suppose we have a sample that includes the solving time vector [18, 19, 300] of three component solvers s_0, s_1 and s_2, with $p = 10$ and $v = 1$, and $\mathfrak{ft} = 19.8$. Assume that the classifier predicts s_1 as the good solver. Since the solving time of s_0 and s_1 are both less than the threshold, we treat that the classifier chooses the correct solver.

D. Learning and Overfitting Mitigation

We run the supervised machine learning of classification in four steps. Firstly, we divide the labeled dataset into training and test data. Secondly, we pick the appropriate kinds of the machine learning algorithms for the classification problem. Thirdly, we use the training data to train the selected classifiers. Finally, we use the test data with the actual labels and the labels predicted by the classifiers to evaluate the performance and get the prediction accuracy. In all these steps, we try to mitigate the over-fitting problem, which is the main concern when using machine learning techniques, to make our portfolio approach more robust and applicable.

Firstly, we apply k-fold cross-validation which is a powerful preventative approach against over-fitting, to divide and train the labeled dataset. Cross-validation allows us to tune hyper-parameters with only the original training dataset. This can keep the test set as a truly unseen dataset for selecting the final model.

When applying the machine learning models, we try to reduce the complexities of all the learning models. Firstly, we choose classic supervised learning models which are simple and use ensembling strategies such as *Bagging* and *Boosting* [16]. Our selected classic supervised learning models are *K-Nearest Neighbor* (*K-NN*) [17], *Logistic Regression* (*LR*) [18], *Support Vector Machine* (*SVM*) [19], *Decision Tree* (*DT*) [20], *Multi-Layer Perceptions* (*MLP*) [21], *AdaBoost* (*ADA*) [22] and *Gradient Boosting* (*GBT*) [23]. We do not use complex deep learning models such as *Convolutional Neural Network* or *Recurrent Neural Network*.

Furthermore, we conduct fine-grained model complexity reduction according to the characteristics of each model. Specifically, for tree-based models such as DT, we limit the maximal tree depth, the maximum number of leaf nodes and the minimum number of samples in a leaf node. We also utilize *post-prune* technology to simplify the tree. For neural network models such as MLP, we only set one hidden layer and ensure that the hidden layer size is smaller than the input layer size. We also use the *early stopping* mechanism, which terminates the training process when the fitness score does not improve any more. For regression-based models such as LR, we make use of *L2 Regularization* to penalize the large coefficients. For K-NN, we set a large K to reduce its sensitivity to noise data.

V. Experimental Setup

To evaluate our portfolio solver, we answer the following research questions:

- RQ1: How does each solver in the SAT competition perform on our dataset? How should we select the component solvers to use in our portfolio solver?
- RQ2: How does the scope affect the performance of each component solvers?
- RQ3: How do the features in each level affect the performance of our portfolio solver? How should we select the feature level?
- RQ4: How much does our portfolio solver improve, compared to each component solver, the random selection approach, and the best selection approach?
- RQ5: How do different machine learning models affect the performance of our portfolio solver?

A. Subjects

As mentioned in Section IV-B1, since different commands of the same model can be modeled as different abstract syntax trees in Alloy, in the experiment, we take commands of Alloy models as our subjects. We only consider the `check` commands and ignore the `run` commands. The reason is that the `run` commands are typically much easier to solve because it mostly does not invoke all formulas in the model. Totally, we collect 119 Alloy models, where 103 models are collected from the sample dataset of Alloy analyzer release 4.2, and 16 models are collected from existing work [12]. Finally, we find 69 `check` command subjects from these models.

B. Metrics

We evaluate the effectiveness of our portfolio approach using the following three metrics. The first one is the accuracy of predicting the fastest solver. The second one is the absolute solving time of our portfolio approach and the baseline approaches. The third metric is the speedup ratio between the portfolio approach and the baseline approaches. Note that we set the thresholds (as introduced in Section IV-C) as $p = 10$ for the relative threshold $\mathfrak{rt}(p)$, and $v = 1$ for the absolute threshold $\mathfrak{at}(v)$.

C. Experimental Protocol

1) Solver Candidates: We select five state-of-the-art SAT solvers from the SAT competition 2018 as our candidate solvers, which is described in Section III.

2) Case Suite: Section IV-A mentioned the potential cases given the number of commands and signatures in each Alloy model, the maximum scope ms and the initial scope is. Here, we set the maximum scope to 40. In addition, since the sample case (typically the CNF file) generation takes time, we also set the timeout of the case generation to 60 seconds. In total, we generated 7376 cases under this timeout and we refer these cases as the whole case suite. We run all 5 candidate solvers over the whole case suite, setting the timeout of SAT solving as 300 seconds. For cases where the solver performs erroneously (like segmentation fault), we set the corresponding solving time as 600 seconds.

Note that we set the p in the relative threshold $\mathfrak{rt}(p)$ to 10%, and the v in the absolute threshold $\mathfrak{at}(v)$ to 1 second, which aims to remove the cases with poor discriminability of solvers' performance. With these two thresholds, we filtered out 3729 cases in which all the candidate solvers have almost no solving time difference. These cases also included the cases in which all the candidate solvers timed out. The remaining case suite contains in total 3647 cases and we refer to this suite as the refined suite. Note that both the whole and refined suites contain 69 Alloy subjects.

3) Platform: We conduct all our experiments on Ubuntu Linux 16.04 with a Intel Core-i7 6700 CPU (3.40 GHz) and 16GB RAM.

TABLE IV: Performance of the Candidate Solvers

Solver		Overall Time(s)	Outperformance		Errors
Name	ID		Num	Percent (%)	
cms55-main-all4fixed	s_0	659590	1997	55	0
Maple_CM	s_1	567402	666	18	3
Maple_CM_ordUIP	s_2	567291	632	17	3
MapleLCMDistChronoBT	s_3	662575	469	13	0
Maple_LCM_Scavel_fix2	s_4	836180	287	8	205

VI. Result Analysis

A. RQ1: Candidate Solver Performance

Table IV depicts the performance of individual candidate solver (labeled by `ID`) on the whole case suite. To be specific, the `Overall Time` presents the overall solving time (in seconds) in all cases, `Outperformance` shows how each solver beats others on the number and percentage of cases, respectively, and `Errors` indicates the number of cases where the solver performs erroneously. From Table IV, we can observe that solver **cms55-main-all4fixed** outperforms on more cases (i.e. 55%) than others. However, interestingly, its overall solving time is only ranked at the third place. This implies that **cms55-main-all4fixed** may lose much in the cases it underperforms. **Maple_CM** and **Maple_CM_ordUIP** shows similar performance in term of all the indicators, where **Maple_CM** performs slightly better in term of the amount of the cases, while **Maple_CM_ordUIP** performs slightly better in term of the overall time. Moreover, despite of the winner in SAT competition, **MapleLCMDistChronoBT** only ranked the fourth place in term of the overall solving time as well as the number of cases. Lastly, **Maple_LCM_Scavel_fix2** performs the worst in terms of all three indicators.

Figure 6 shows the similarity in each combination of two candidate solvers (denoted by the corresponding solver ID), where the similarity is determined with the relative threshold $\mathfrak{rt}(p)$ and the absolute threshold $\mathfrak{at}(v)$. We set two pairs of the thresholds for the similarity analysis. One is $p = 10$ and $v = 1$; the other one is $p = 20$ and $v = 2$. The similarity is the percentage of the cases of which the solving time is within the determined thresholds. We can observe from Figure 6 that, **Maple_CM** and **Maple_CM_ordUIP** shows the highest similarity which is also consistent with the observation from Table IV. This may be rooted in the fact that **Maple_CM_ordUIP** is developed based on **Maple_CM** with a modified learning strategy. For details, please refer to the Proceedings of SAT Competition 2018 [24].

We decide to eliminate **Maple_LCM_Scavel_fix2** from our candidate solver list, given the quite amount of cases in which it shows erroneous behavior and the few amount of win cases it can potentially contribute (e.g. it only beats others on 287 cases while crashes on 205 cases). In addition, since **Maple_CM** and **Maple_CM_ordUIP** show similar solving behavior, we only select one of them. We assume **Maple_CM** may have slightly more possibility to contribute to the portfolio approach, since it wins in more cases. Therefore, we choose **Maple_CM** over **Maple_CM_ordUIP**.

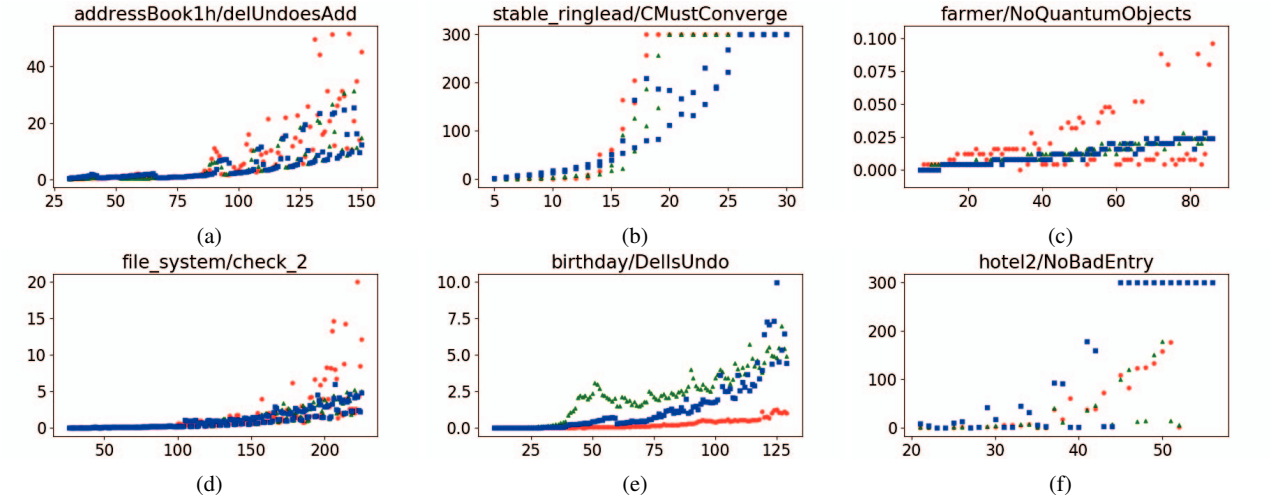

Fig. 5: Solving Behaviors with Scopes (x axis indicates the solving time in seconds; y axis indicates the total scope of all the signatures in the corresponding Alloy model)

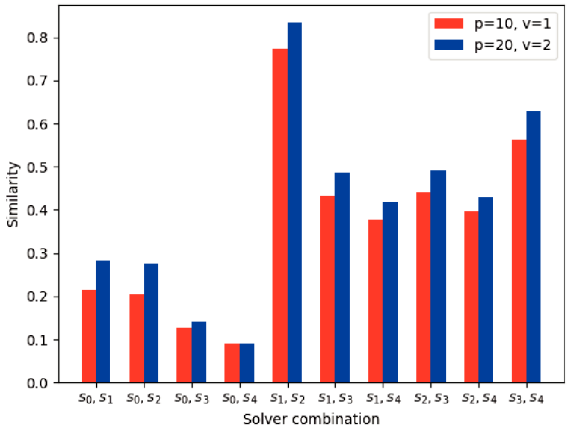

Fig. 6: Similarity in each combination of two candidate solvers

Finally, the SAT solvers, namely **MapleLCMDistChronoBT**, **Maple_CM** and **cms55-main-all4fixed** are selected as the component solvers for our portfolio approach, and we label them as c_0, c_1 and c_2 respectively.

B. RQ2: Solving Behaviors with Scope

This section investigates how the scope affects the performance of the component solvers. Figure 5 presents their performances on 6 Alloy subjects, in which the x axis indicates the total scope of all the signatures in the corresponding Alloy model, the y axis indicates the solving time in seconds. The red, green and blue dots indicate solver c_0, c_1 and c_2, respectively. According to Figure 5, we can find that as the scope increases, the solving time of all component solvers increases generally. However, there is no obvious patterns behind the curves to help us predict which solver performs the best in which kind of Alloy subject. This makes machine learning highly demanding for classifying the component solvers according to their suitability for the input Alloy subjects.

C. RQ3: Feature Importance and Selection

The length of the relational bound feature vector we introduced in Section IV-B2 is set to 40. Given this, there are 208 features in all the three levels. Among these, 45 are constant across all cases. After removing these features, we have 85 features in level 1, 75 features in level 2, and 3 features in level 3. In order to evaluate how the features in each level affect the model prediction, we conduct an ablation study of all the combinations of the feature levels. Table V shows the prediction accuracy in each machine learning model under each combination.

We can see that the features from level 2 contribute the most in the prediction results. In all the machine learning models except for MLP, even using the level 2 features alone outperforms the circumstances when using the features from all the three levels. The level 1 features contribute more than the level 3 features. Using level 3 features combined with level 1 can make improvements compared with using the level 1 features alone; While using level 3 features combined with level 2 can downgrade the performance compared with using the level 2 features alone. Furthermore, using combination of level 2 and 3 features outperforms using combination of level 1 and 3 features. These two observations indicates that features in level 1 are not as good as the features in level 2. In addition, level 2 features can almost cover the information expressed in level 3 features. This result can also be explained with the translation in Alloy analyzer, that the optimized and simplified Kodkod relational formulas are closer to the boolean formulas the SAT solver directly uses. Additionally, we can extract more features at level 2 compared with level 3, which means level

TABLE V: Model accuracy on different feature combinations

Model	Accuracy(%)						
	(1)	(2)	(3)	(1, 2)	(1, 3)	(2, 3)	(1, 2, 3)
svm	73.55	80.26	63.02	76.20	74.76	79.45	76.95
decisiontree	70.35	80.53	69.29	75.42	72.81	77.59	75.91
adaboost	73.70	81.68	71.08	73.78	74.27	81.33	75.73
gradientboosting	77.27	81.43	70.04	78.88	74.42	80.37	78.28
bagging	73.76	81.09	72.05	80.46	73.39	81.25	80.50
knn	76.46	80.92	71.02	80.14	76.66	80.89	80.14
lr	75.28	77.76	59.51	75.08	74.70	76.55	76.63
lsvm	72.39	76.52	59.42	73.69	72.33	75.86	74.18
mlp	75.67	77.27	71.89	78.33	77.36	76.13	78.67

TABLE VI: Portfolio Approach Performance (time in seconds)

	Portfolio	c_0	c_1	c_2	BS	RS
solve time	64638.5	176872.0	84627.5	180407.4	40046.5	147225.8
ratio	NA	2.74	1.31	2.79	0.62	2.28

2 features can provide more accurate and expressive logical information.

Given the above discussion, we decide to only select the features from level 2 for our portfolio approach. The feature extraction time in level 2 for the whole case suite is 119.15 seconds, which is on average **0.016** seconds for each case.

D. RQ4: Effectiveness of Portfolio Approach

Our portfolio approach applies Adaboost classification algorithm to do the solver selection. The accuracy of the selection is 82%. We conduct 10-fold cross-validation to evaluate our portfolio approach. Note that we divide our refined case suite into 10 groups in unit of Alloy subjects. The solving time of the portfolio approach is based on the prediction of each validation set where the Alloy subjects are unseen in the corresponding training set. For instance, if the prediction on a sample in validation set is component solver c_0, we get the solving time of c_0 as the solving time of our portfolio approach on this sample. Since there is no overlap among the validation sets, we can get all the predicted solving time of the case suite. We repeat the 10-fold cross-validation for 10 times to mitigate the over-fitting problem.

The baselines of our portfolio approach are the individual component solvers (c_0, c_1 and c_2), the virtual solver which randomly selects the component solver (RS), and the virtual solver which always select the best component solver (BS). The results of portfolio approach comparing with the baselines are shown in Table VI in which `solve time` means the total solving time of each solver on the refined test suite, and `ratio` means the ratio between the baseline time overhead and our portfolio approach time overhead. We can conclude from the results that our portfolio approach outperforms each individual component solver, as well as the random selection approach significantly. In addition, the speed of our portfolio approach achieves 62% of best selection approach speed and saves 19989 seconds comparing to the best on average component solver c_1.

E. RQ5: Machine Learning Model Effectiveness

We tried 9 machine learning models and the results are shown in Table VII, in which `mean` and `std` indicate the average predicted solving time and the standard deviation in 10 repetitions, `Acc` means the accuracy with the threshold consideration. According to the results, we can say that all the machine learning models perform relatively similar and stable regarding the average solving time as well as the standard deviation of the prediction. Furthermore, they all outperform the three individual component solvers as well as the virtual solver which does the random selection. Besides, these results can further confirm that our extracted features are not only informative but also robust.

Note that Decision tree, AdaBoost and SVM all perform well in terms of the predicted time. The reason we choose AdaBoost over Decision Tree and SVM is because it achieves both the accuracy (in terms of the mean predicted time and the accuracy) and the stability (in terms of the standard deviation) of prediction.

VII. Related Work

Our portfolio solver helps solve the problem of choosing a good solver for given Alloy models based on machine learning approach. Here, we discuss the related works in portfolio combinatorial solvers using machine learning and Alloy.

A. Portfolio CSP Solvers Using Machine Learning

CPHydra [11] is the first portfolio CSP solver which applies a machine learning technique called k-nearest neighbor algorithm (K-NN) to exploit the similarity of problems and select the component solvers. The extracted features include both static (syntactic) and dynamic features. The dynamic features include modeling choices and search statistics. CPHydra combines machine learning with the idea of partitioning CPU time between the component solvers to select the solvers and maximize the expected number of solved problems within a settled time limit. SUNNY [29] is a lazy portfolio approach which uses the K-NN algorithm just as CPHydra, but applies three heuristics to decide the order of the component solvers to run and minimize the average solving time of each problem.

TABLE VII: Execution time of predicted portfolio and acceleration ratio

Model	Predicted time		Acc(%)	Acceleration ratio					
	mean	std(%)		l.t.	s.t.	a.t.	s_0	s_1	s_2
svm	64938.44	0.00	0.80	4.55	0.62	2.27	2.72	1.30	2.78
decisiontree	64477.84	4.03	0.81	4.59	0.62	2.29	2.75	1.31	2.80
adaboost	64638.51	0.10	0.82	4.57	0.62	2.28	2.74	1.31	2.79
gradientboosting	70040.82	0.00	0.81	4.21	0.57	2.10	2.53	1.21	2.58
bagging	69628.43	0.40	0.81	4.24	0.58	2.11	2.54	1.22	2.59
knn	76148.18	0.00	0.81	3.88	0.53	1.94	2.32	1.11	2.37
lr	76748.25	0.00	0.78	3.85	0.52	1.92	2.30	1.10	2.35
lsvm	75977.08	1.50	0.77	3.89	0.53	1.94	2.33	1.11	2.38
mlp	70642.10	1.70	0.77	4.18	0.57	2.09	2.50	1.20	2.55

Variants of SUNNY have been proposed – a sequential portfolio solver called sunny-cp [30], and a parallel solver called sunny-cp2 [31]. In addition, Stojadinović et al. [32] propose a simplified K-NN based portfolio solver which has a short training phase and achieves better performance.

Some researchers have looked at the problem from other angles. Loreggia et al. [33] introduce an automated way for generating features by training a neural network on images translated from problems. Arbelaez et al. [34], [35] use support vector machines (SVM) to dynamically adapt the search heuristics inside a single CSP solver. Stojadinović et al. [36] and Hurley et al. [37] propose portfolio CSP approaches for selecting among different SAT encoding, instead of CSP solvers. An empirical study of the portfolio approaches for CSPs is presented by Amadini [38], [39].

B. Portfolio SAT and SMT Solvers Using Machine Learning

SATzilla-07 [40] is the first mature SAT portfolio solver which selects solvers using machine learning models for runtime prediction. SATzilla [41] performs better than SATzilla-07 and becomes a successful approach, making the portfolio construction scalable and completely automated. To achieve that, it integrates local-search solvers as component solvers and applies *hierarchical* machine learning models on different types of SAT problems. Malitsky et al. [42] investigated alternative ways of building algorithm portfolios with K-NN classification to determine which solver to use for a given problem. In the SMT literature, Abdul Aziz et al. [43] uses a linear machine learning technique called Ridge regression to estimate the hardness of SMT problems. A Portfolio bit-vector SMT solver called Wombit [44] applies a Decision Tree model to select the candidate solvers.

Note that the potential advantage of using our portfolio solver instead of the off-the-shelf portfolio SAT solver is that our solver can make use of the features from relational logic which makes our approach more specific and targeted for Alloy models. Since the component SAT solvers we applied in our portfolio solver are totally different from the ones in the of-the-shelf portfolio solver, we leave an apple-to-apple comparison for future work.

C. Alloy

Over the past years, researchers have developed many extensions for Alloy [45]–[47]. Alloy* [48] allows users to write models in second order logic. AUnit [49] defines unit testing for Alloy. MuAlloy [50], [51] brings mutation testing to Alloy. ASketch [52]–[54] is able to sketch partial Alloy models. AlloyFL [55] helps to locate faults in Alloy models.

VIII. Thread to Validity

Threats to internal validity are about whether over-fitting may have occurred in the experimental evaluation, that is, whether the generated machine learning model is designed to fit the training data so closely that it becomes inaccurate for unseen data. If the over-fitting happens, then our conclusions about the advantages of the portfolio approach may not remain valid once the approach is applied more broadly. To mitigate this, the 10-fold cross validation has been used. Besides, the techniques for reducing the machine learning model complexity has also been applied to mitigate the over-fitting risk.

The main threat to external validity is that our collected Alloy models may not generalize to other unseen models. We use the models from the examples in Alloy Analyzer tool as our subjects, but these models may not be representative of other Alloy models. Although the models are from a diversity of sources and applications, it is still possible that they exhibit an undesirable lack of variety. In particular, previous research has shown that machine learning techniques may behave differently on a totally different problem.

IX. Future Work & Conclusion

Regarding to the above threats, we plan to generate a more pervasive Alloy model dataset from the real-world systems to make our portfolio approach more robust and applicable.

This paper proposed a portfolio approach for the Alloy Analyzer based on machine learning techniques which automatically selects an appropriate SAT solver for a certain Alloy model. To achieve this, we extract the Alloy specific features from three levels: Alloy source code level, Kodkod formula level and the boolean formula level. Experimental results show that our portfolio approach outperforms each of the component solvers as well as random solver selection approach.

Acknowledgment

We thank Sasa Misailovic for helpful discussion and the anonymous reviewers for valuable comments. This research was partially supported by the US National Science Foundation under Grant No. CCF-1718903.

REFERENCES

[1] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: A notation for detailed design," in *Behavioral Specifications of Businesses and Systems*, 1999.

[2] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *ASE*, 2001.

[3] D. Jackson, "Alloy: A lightweight object modelling notation," *TOSEM*, 2002.

[4] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, 2003.

[5] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The margrave tool for firewall analysis," in *LISA*, 2010.

[6] N. Ruchansky and D. Proserpio, "A (not) nice way to verify the openflow switch specification: Formal modelling of the openflow switch using alloy," *SIGCOMM*, 2013.

[7] S. Maoz, J. O. Ringert, and B. Rumpe, "Cd2alloy: Class diagrams analysis using alloy revisited," in *MODELS*, 2011.

[8] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *TACAS*, 2007.

[9] "Sat competition 2018 homepage," 2018. [Online]. Available: http://sat2018.forsyte.tuwien.ac.at/

[10] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, April 1997.

[11] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan, "Using case-based reasoning in an algorithm portfolio for constraint solving," in *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008, pp. 210–216.

[12] K. Wang, A. Sullivan, and S. Khurshid, "Automated model repair for alloy," in *ASE*, 2018.

[13] K. Wang, A. Sullivan, and S. Khurshid, "Arepair: A repair framework for alloy," in *ICSE*, 2019.

[14] "Alloy 4 tutorial materials." [Online]. Available: http://alloy.lcs.mit.edu/alloy/tutorials/day-course/

[15] E. Torlak, "Kodkod documentation." [Online]. Available: http://emina.github.io/kodkod/doc/

[16] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction (Second Edition)*. Springer, 2017, vol. 1.

[17] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992. [Online]. Available: https://amstat.tandfonline.com/doi/abs/10.1080/00031305.1992.10475879

[18] P. McCullagh and J. Nelder, *Generalized Linear Models, Second Edition*, ser. Chapman and Hall/CRC Monographs on Statistics and Applied Probability Series. Chapman & Hall, 1989. [Online]. Available: http://books.google.com/books?id=h9kFH2_FfBkC

[19] M. A. Hearst, "Support vector machines," *IEEE Intelligent Systems*, vol. 13, no. 4, pp. 18–28, Jul. 1998. [Online]. Available: http://dx.doi.org/10.1109/5254.708428

[20] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986. [Online]. Available: http://dx.doi.org/10.1023/A:1022643204877

[21] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[22] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[23] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[24] M. J. H. Heule, M. J. J´arvisalo, and M. Suda, "Proceedings of sat competition 2018: Solver and benchmark descriptions," ser. Department of Computer Science Series of Publications B, 2018. [Online]. Available: http://hdl.handle.net/10138/237063

[25] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: The state of the art," 2012, coRR, http://arxiv.org/abs/1211.0906.

[26] L. Kotthoff, "Algorithm selection for combinatorial search problems: A survey," *AI Magazine*, vol. 35, no. 3, pp. 48–60, 2014.

[27] K. A. Smith-Miles, "Cross-disciplinary perspectives on meta-learning for algorithm selection," *ACM Computing Surveys*, vol. 41, no. 1, pp. 6:1–6:25, 2009.

[28] L. Kotthoff, I. P. Gent, and I. Miguel, "An evaluation of machine learning in algorithm selection for search problems," *AI Commun.*, vol. 25, no. 3, pp. 257–270, Aug. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2350296.2350300

[29] R. Amadini, M. Gabbrielli, and J. Mauro, "SUNNY: A lazy portfolio approach for constraint solving," *Theory and Practice of Logical Programming*, vol. 14, no. 4–5, pp. 509–524, 2014.

[30] R. Amadini, M. Gabbrielli, and J. Mauro, "SUNNY-CP: A sequential CP portfolio solver," pp. 1861–1867, 2015.

[31] R. Amadini, M. Gabbrielli, and J. Mauro, "A multicore tool for constraint solving," pp. 232–238, 2015.

[32] M. Stojadinović, M. Nikolić, and F. Marić, "Short portfolio training for CSP solving," 2015, coRR, https://arxiv.org/abs/1505.02070.

[33] A. Loreggia, Y. Malitsky, H. Samulowitz, and V. A. Saraswat, "Deep learning for algorithm portfolios," in *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, 2016, pp. 1280–1286.

[34] A. Arbelaez, Y. Hamadi, and M. Sebag, "Online heuristic selection in constraint programming," in *Proceedings of the International Symposium on Combinatorial Search*, 2009, https://hal.inria.fr/inria-00392752/.

[35] A. Arbelaez, Y. Hamadi, and M. Sebag, "Continuous search in constraint programming," in *Autonomous Search*, Y. Hamadi *et al.*, Eds., 2011, ch. 9, pp. 219–243.

[36] M. Stojadinović and F. Marić, "meSAT: Multiple encodings of CSP to SAT," *Constraints*, vol. 19, no. 4, pp. 380–403, 2014.

[37] B. Hurley, L. Kotthoff, Y. Malitsky, and B. O'Sullivan, "Proteus: A hierarchical portfolio of solvers and transformations," in *Integration of AI and OR Techniques in Constraint Programming: Proceedings of the 11th International Conference (CPAIOR'14)*, H. Simonis, Ed., vol. 8451, 2014, pp. 301–317.

[38] R. Amadini, M. Gabbrielli, and J. Mauro, "An extensive evaluation of portfolio approaches for constraint satisfaction problems," *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 3, no. 7, pp. 81–86, 2016.

[39] R. Amadini, M. Gabbrielli, and J. Mauro, "An empirical evaluation of portfolios approaches for solving CSPs," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: Proceedings of the 10th International Conference*, C. Gomes and M. Sellmann, Eds., 2013, pp. 316–324.

[40] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla-07: The design and analysis of an algorithm portfolio for sat," in *Principles and Practice of Constraint Programming – CP 2007*, C. Bessière, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 712–727.

[41] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: Portfolio-based algorithm selection for SAT," *CoRR*, vol. abs/1111.2249, 2011. [Online]. Available: http://arxiv.org/abs/1111.2249

[42] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, "Non-model-based algorithm portfolios for sat," in *Theory and Applications of Satisfiability Testing - SAT 2011*, K. A. Sakallah and L. Simon, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 369–370.

[43] M. A. Aziz, A. Wassal, and N. Darwish, "A machine learning technique for hardness estimation of QFBV SMT problems," in *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories (SMT'12)*, ser. EPiC Series in Computing, P. Fontaine and A. Goel, Eds., vol. 20. EasyChair, 2013, pp. 57–66.

[44] W. Wang, H. Søndergaard, and P. J. Stuckey, "Wombit: A portfolio bit-vector solver using word-level propagation," *Journal of Automated Reasoning*, Nov 2018. [Online]. Available: https://doi.org/10.1007/s10817-018-9493-1

[45] A. Sullivan, K. Wang, S. Khurshid, and D. Marinov, "Evaluating state modeling techniques in Alloy," in *SQAMIA*, 2017.

[46] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Aluminum: principled scenario exploration through minimality," in *ICSE*, 2013.

[47] T. Nelson, N. Danas, D. J. Dougherty, and S. Krishnamurthi, "The power of "why" and "why not": Enriching scenario exploration with provenance," in *FSE*, 2017.

[48] A. Milicevic, J. P. Near, E. Kang, and D. Jackson, "Alloy*: A general-purpose higher-order relational constraint solver," in *ICSE*, 2015.

[49] A. Sullivan, K. Wang, and S. Khurshid, "AUnit: A Test Automation Tool for Alloy," in *ICST*, 2018.

[50] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid, "Automated test generation and mutation testing for Alloy," in *ICST*, 2017.

[51] K. Wang, A. Sullivan, and S. Khurshid, "MuAlloy: A Mutation Testing Framework for Alloy," in *ICSE*, 2018.

[52] K. Wang, A. Sullivan, M. Koukoutos, D. Marinov, and S. Khurshid, "Systematic generation of non-equivalent expressions for relational algebra," in *ABZ*, 2018.
[53] K. Wang, A. Sullivan, D. Marinov, and S. Khurshid, "Solver-based sketching Alloy models using test valuations," in *ABZ*, 2018.
[54] K. Wang, A. Sullivan, D. Marinov, and S. Khurshid, "Asketch: A sketching framework for alloy," in *FSE*, 2018.
[55] K. Wang, A. Sullivan, D. Marinov, and S. Khurshid, "Fault localization for declarative models in Alloy," in *eprint arXiv:1807.08707*, 2018.