Proc. ISoLA 2018, © Springer

Strategy Selection for Software Verification Based on Boolean Features

A Simple but Effective Approach

Dirk Beyer and Matthias Dangl

LMU Munich, Germany

Abstract. Software verification is the concept of determining, given an input program and a specification, whether the input program satisfies the specification or not. There are different strategies that can be used to approach the problem of software verification, but, according to comparative evaluations, none of the known strategies is superior over the others. Therefore, many tools for software verification leave the choice of which strategy to use up to the user, which is problematic because the user might not be an expert on strategy selection. In the past, several learning-based approaches were proposed in order to perform the strategy selection automatically. This automatic choice can be formalized by a strategy selector, which is a function that takes as input a model of the given program, and assigns a verification strategy. The goal of this paper is to identify a small set of program features that (1) can be statically determined for each input program in an efficient way and (2) sufficiently distinguishes the input programs such that a strategy selector for picking a particular verification strategy can be defined that outperforms every constant strategy selector. Our results can be used as a baseline for future comparisons, because our strategy selector is simple and easy to understand, while still powerful enough to outperform the individual strategies. We evaluate our feature set and strategy selector on a large set of 5 687 verification tasks and provide a replication package for comparative evaluation.

Keywords: Strategy Selection, Software Verification, Algorithm Selection, Program Analysis, Model Checking

1 Introduction

The area of automatic software verification is a mature research area, with a large potential for adoption in industrial development practice. However, there are many usability issues that hinder the widespread use of the technology that is developed by researchers. One of the usability problems is that it is not explainable, for a given input program, which verification strategy to use. Different verification tools, algorithms, abstract domains, configurations, coexist with their different strengths in terms of approaching a verification problem.

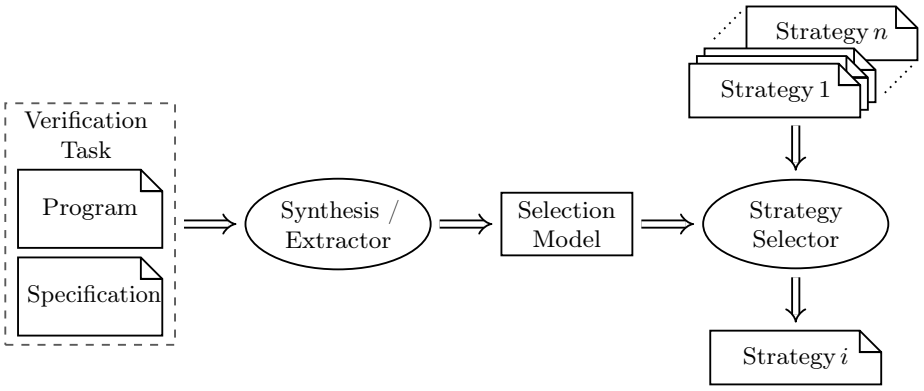


Fig. 1: Architecture of strategy selection (compare with Fig. 3 by Rice [28])

The insight that different verification techniques have different strengths was emphasized several times in the literature already. Most intensively, this can be derived from the results of the competition on software verification [4].[1] A recent survey on SMT-based algorithms [6] (including bounded model checking, k-induction, predicate abstraction, and IMPACT) explains this insight concretely on specific example programs (from different categories of a well-known benchmark repository[2]): For each of the four considered algorithms, one example program is given that only this algorithm can efficiently verify and all other algorithms fail or timeout on this program. While there are powerful basic techniques, combinations or a selection are often a valuable strategy to further improve.

The problem has been understood for a long time in the research community, and there are several methods to approach the problem [24]. The standard techniques are sequential and parallel combinations [2, 7, 23, 26, 35]. These techniques are mostly based on statically assembling the combinations, and, by trying out one technique after the other (sequential) or by trying all at the same time (parallel), the problem is often solved by the approach that works best. However, there might be a considerable amount of resources wasted on unsuccessful computation work. For example, it might happen that one approach could solve the problem if all available resources were given to it, but since the resource is shared and assigned to several approaches, the overall verification does not succeed.

In order to solve this problem, a few techniques were proposed in the last few years that automatically select a potentially good verification strategy based on machine learning [18, 19, 20, 28, 32]. All those proposals share the common idea of strategy selection.

Strategy selection can be illustrated by the flow diagram in Fig. 1: The verification task (consisting of the source code of the input program and the

[1] `https://sv-comp.sosy-lab.org/2018/results/results-verified/`
[2] `https://github.com/sosy-lab/sv-benchmarks`

specification) is first analyzed and an abstract selection model is constructed (synthesized or extracted). The strategy selector predicts a strategy (from a given set of strategies) that should be used to solve the verification task, based on the information in the selection model.

The *selection model* can be either a vector of feature values, as defined by Rice's 'feature space' [28] (and implemented for software verification by, e.g., [19, 20, 32]), a graph representation of the program (e.g., [18]), or some other characteristics of the program and its specification. A selection model is *useful* if it contains sufficient information to distinguish verification tasks that need to be verified with different strategies. The model construction phase needs to extract the information from the source code of the input program and the specification. For example, the values of a feature vector might be extracted by static source-code measures.

The *set of strategies* (also called 'algorithm space' [28]) is either a set of algorithms, verification tools, different configurations of a configurable verification framework, or just a mere set of different parameter specifications for a single verifier.

The *strategy selector* is a function that takes as input a set of strategies and the selection model that represents some information about the program and its specification, and returns as output the strategy that is predicted to be useful to solve the verification task that is represented by the selection model.

Contributions. This paper makes the following contributions:

- We define a minimalist selection model, which (1) consists of an extremely small set of features that define the selection model and (2) a minimal range of values: all features are of type Boolean.
- We define an extremely simple strategy selector, which is based on insights from verification researchers.
- We implemented our feature measures and strategy selection in CPAchecker; the replication package contains all data for replicating the results.
- We perform a thorough experimental evaluation on a large benchmark set.

Related Work. We categorize the related work into the three areas of combinations, models, and machine learning.

Sequential and Parallel Combinations (Portfolios). While it seems obvious that combinations of strategies have a large potential, the topic was not yet systematically investigated in the area of software verification, while it has been used in other areas for many years [24, 28]. One of the first ideas to combine different tools was for eliminating false alarms: after the core verifier has found an error path, this error path is not immediately reported to the user, but first converted into a program again which is then verified by an external verifier, and only if that external tool reports an error path as well, then the alarm is shown as a result to the user. [3]

[3] An early version of CPAchecker [9] had constructed a path program [8], dumped it to a file in C syntax, and then called Cbmc [17] as external verifier for validation. Meanwhile, such an error-path check is a standard component in many verifiers.

Other examples for sequential combinations are CPAchecker and SDV. CPAchecker [9] won the competition on software verification 2013 (SV-COMP'13, [3]) using a sequential combination [35] that started with explicit-state model checking for up to 100 s and then switched to a predicate analysis [10]. The static driver verification (SDV) [2] tool chain at Microsoft used a sequential combination (described in [32]) which first runs Corral [25] for up to 1 400 s and then Yogi [27].

Examples of parallel combinations are the verifiers Ufo [23] and PredatorHP [26], which start several different strategies simultaneously and take the result from the approach that terminates first.

Conditional model checking [7] is a technique to construct combinations with passing information from one verifier to the other. This technique can also be used to split programs into parts that can be independently verified [30].

Selection Models. A strategy selector needs a selection model of the program, in order to be able to classify the program and select a strategy. The classic way of abstracting is to define a set of features and the resulting vector of feature values is the selection model, which is in turn given to the strategy selector as input. There are various works on identifying features that are useful for classifying programs using its source code. Domain types [1] refine the integer types of C programs into more fine-grained integer types, in order to estimate what kind of abstract domain should be used to verify the program, for example, whether a BDD-based analysis or an SMT-based analysis is preferable. Variable roles [15, 29, 33, 34] were used to analyze and understand programs, but also to classify program variables [22] according to how they are used in the program, i.e., what their role is. It has been shown that variable roles can help to determine which predicates should be used for predicate abstraction [21]. More sophisticated selection models can be used for machine-learning-based approaches. For example, one approach is based on graph representations of the program [18].

Machine-Learning-Based Approaches. The technique MUX [32] can be used to synthesize a strategy selector for a set of features of the input program and a given number of strategies. The strategies are verification tools in this case, and the feature values are statically extracted from the source code of the input program. Unfortunately, this technique is not reproducible, as reported by others [20]. Later, a technique that uses more sophisticated features was proposed [19, 20]. While the above techniques use explicit features (defined by measures on the source code), a more recently developed technique [18] leaves it up to the machine learning to obtain insights from the input program. The advantage is that there is no need to define the features: the learner is given the control-flow graph, the data-dependency graph, and the abstract syntax tree, and automatically derives internally the characteristics that it needs. Also, the technique predicts a ranking, that is, the strategy selector is a function that maps verification tasks not to a single strategy, but to a sequence of strategies.

2 An Approach Based on Simple Boolean Features

Our goal is to define a strategy-selection approach that is simple and easy to understand but still effectively improves the overall performance.

2.1 Selection Model

We identify the following criteria from which we define our selection model:

- The model is based on features of the input program that are efficiently extractable from the program's source code using a simple static analysis.
- The model consists of a small set of features.
- The features have a small set of values.

Based on sets of program characteristics that were reported in the literature [1, 22], we selected a few extremely coarse features. We will later evaluate whether our choice of features can instantiate a model that contains sufficient information to distinguish programs that should be verified by different strategies. Let $V = P \times S$ be the set of all verification tasks, each of which consists of a program from the set P and a specification from the set S, and let $\mathbb{B}$ be the set of Boolean values. We define the following four features for our selection model:

hasLoop : $V \to \mathbb{B}$ with
$\mathsf{hasLoop}((p, \cdot)) = \mathit{true}$ if program p has a loop, and *false* otherwise

hasFloat : $V \to \mathbb{B}$ with
$\mathsf{hasFloat}((p, \cdot)) = \mathit{true}$ if program p has a variable of a floating-point type (`float`, `double`, and `long double` in C), and *false* otherwise

hasArray : $V \to \mathbb{B}$ with
$\mathsf{hasArray}((p, \cdot)) = \mathit{true}$ if program p has a variable of an array type, and *false* otherwise

hasComposite : $V \to \mathbb{B}$ with
$\mathsf{hasComposite}((p, \cdot)) = \mathit{true}$ if program p has a variable of a composite type (`struct` and `union` in C), and *false* otherwise

For example, consider a program with a loop and only variables of integer type; the selection model would be the feature vector $(\mathit{true}, \mathit{false}, \mathit{false}, \mathit{false})$.

2.2 Strategies

For our example instantiation of a strategy-selection approach, we use different strategies from one verification framework. [4] We choose the software-verification framework CPAchecker as framework to configure our strategies, because it

[4] This has the advantage that the performance difference is not caused by the use of different programming languages, parser frontends, SMT solvers, libraries, but by the conceptual difference of the strategy (better internal validity). While it would be technically easy to extend the set of available strategies to other software verifiers, we already obtain promising results by just using different CPAchecker strategies.

consistently yielded good results in the competition on software verification (SV-COMP) [4], and we can actually also compare against CPA-Seq, the winning strategy that CPAchecker used in SV-COMP 2018. [5] Also, CPAchecker is highly configurable and provides a comprehensive set of algorithms and components to choose from (e.g., [6, 11]) as well as a simple mechanism for sequential [35] and parallel composition [31]. The description of our three verification strategies will refer to the following components: [6]

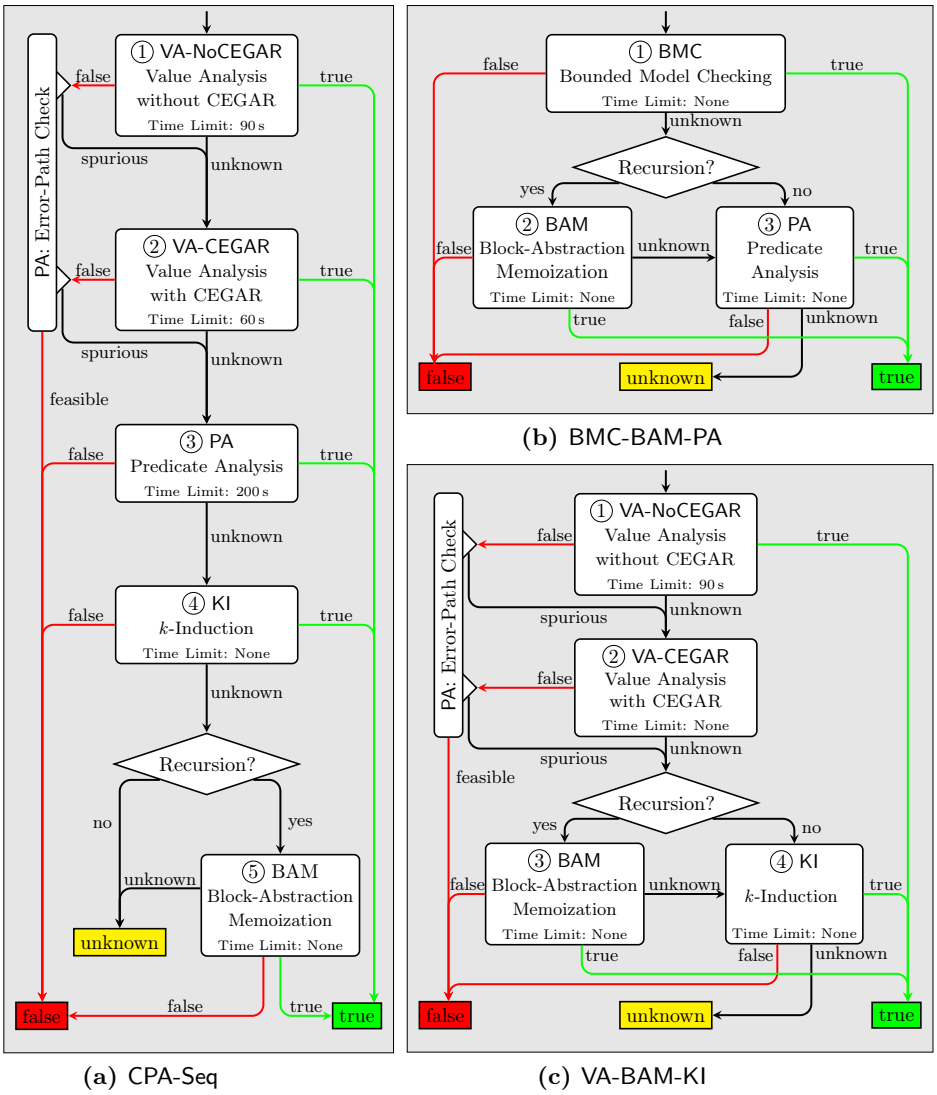
VA-NoCEGAR: value analysis without CEGAR [7] [11]
VA-CEGAR: value analysis with CEGAR [11]
PA: predicate analysis with CEGAR [10]
KI: k-induction with continuously refined invariant generation [5]
BAM: block-abstraction memoization (BAM) [36] for a composite abstract domain of predicate analysis and value analysis
BMC: bounded model checking (BMC) [13]

The set of three verification strategies that we use in our strategy selector are the above mentioned strategy CPA-Seq that won the last competition and two more strategies that are based on components from the above list:

CPA-Seq is a sequential combination of VA-NoCEGAR, VA-CEGAR, PA, KI, and BAM as depicted in Fig. 2a: VA-NoCEGAR runs for up to 90 s, then VA-CEGAR runs for up to 60 s, then PA for up to 200 s, followed by KI for the remaining time. Any of the components may terminate early if it detects that it cannot handle the task. If none of the aforementioned components can handle the task and the last one (KI) fails because the task requires handling of recursion, the BAM component runs, which in our implementation is the only one that is able to handle recursion but lacks support for handling pointer aliasing and is therefore only desirable as a fallback for recursive tasks. If either VA-NoCEGAR or VA-CEGAR find a bug in the verification task, the error path is checked for feasibility with a PA-based error-path check; if the check passes, the bug is reported, otherwise, the component result is ignored and the next component runs.

BMC-BAM-PA is a sequential combination of BMC, BAM, and PA as depicted in Fig. 2b. As above, any of the components may terminate early if it detects that it cannot handle the task; otherwise there are no individual time limits for components in this strategy: As a result the first component of this strategy, BMC, runs until it solves the task or fails. If it fails because the task requires handling of recursion, the BAM component runs, with the same reasoning as for CPA-Seq; if the reason why bounded model checking failed was not recursion or if BAM also fails to solve the task, PA runs. This means that BAM and PA are only used as fallback components if the BMC component fails due to recursion or some other unsupported feature, whereas in all other cases, BMC would be the only component that runs.

[5] `https://sv-comp.sosy-lab.org/2018/`
[6] KI is, strictly speaking, already a composition, because it uses bounded model checking (BMC) [14] as a component.
[7] CEGAR is the abbreviation for counterexample-guided abstraction refinement [16].



(a) CPA-Seq

(b) BMC-BAM-PA

(c) VA-BAM-KI

Fig. 2: Sequential combinations of strategies

VA-BAM-KI is a sequential combination of VA-NoCEGAR, VA-CEGAR, BAM, and KI, as depicted in Fig. 2c. As above, any of the components may terminate early if it detects that it cannot handle the task; only the first component, VA-NoCEGAR, has an individual time limit and runs for up to 90 s. Afterwards, VA-CEGAR runs until it exceeds its time limit, fails, or solves the task. As in CPA-Seq, if either VA-NoCEGAR or VA-CEGAR find a bug in the verification task, the error path is checked for feasibility with a PA-based error-path

check; if the check passes, the bug is reported, otherwise, the component result is ignored and the next component runs. If VA-CEGAR fails because the task requires handling of recursion, the BAM component runs, with the same reasoning as for CPA-Seq; if the reason why VA-CEGAR failed was not recursion or if BAM also fails to solve the task, KI runs. This means that BAM and KI are only used as fallback components if VA-NoCEGAR and VA-CEGAR both fail due to recursion or some other unsupported feature, whereas in all other cases, either VA-NoCEGAR would solve the task within at most 90 s, or VA-CEGAR would attempt to solve it in the remaining time without switching to any further components.

2.3 Strategy Selector

Based on the three strategies and the selection model described above, we define our strategy selector Model-Based. Our strategy selector chooses the strategy based on the selection model as follows: It is defined to always choose the strategy BMC-BAM-PA if hasLoop is *false*, because if there is no loop, we do not need any potentially expensive invariant-generating algorithm. If hasLoop is *true*, and either of hasArray, hasFloat, or hasComposite is *true*, it chooses the strategy VA-BAM-KI. If hasLoop is *true* and all of hasArray, hasFloat, and hasComposite are *false*, it chooses the strategy CPA-Seq:

$$\text{strategy} = \begin{cases} \text{BMC-BAM-PA} & \text{if } \neg\text{hasLoop} \\ \text{VA-BAM-KI} & \text{if hasloop} \wedge (\text{hasFloat} \vee \text{hasArray} \vee \text{hasComposite}) \\ \text{CPA-Seq} & \text{otherwise} \end{cases}$$

While CPA-Seq consists of a wider variety of components that should in theory be more accurate for these complex features, VA-BAM-KI, which consists mainly of value analysis, does not require expensive SMT solving and therefore often solves tasks where CPA-Seq exceeds the resource limitations.

3 Evaluation

In this section, we present an experimental study to compare the effectiveness of our approach to strategy selection to various fixed strategies (i.e., constant strategy selectors) and to serve as a baseline for future comparisons of potentially more elaborate approaches.

3.1 Evaluation Goals

The goal of our experimental evaluation is to confirm the following claims:

Claim 1: We claim that combining different strategies sequentially is more effective than each individual strategy by itself. To confirm this claim, we evaluate the composite strategy CPA-Seq as well as each of its individual components, and compare their results. For a successful confirmation, CPA-Seq must yield a higher score than each of its component strategies. If confirmed, this claim supports the insight that combinations should be used in practice.

Claim 2: We claim that by classifying a verification task using a small set of features and selecting a strategy to solve a task from a small set of verification strategies based on this classification, we can further improve effectiveness significantly. To confirm this claim, we evaluate three verification strategies individually, as well as two strategy selectors that can choose from the three sequential strategies: One of the strategy selectors will choose randomly, while the other one will base its choice on the selection model that we extracted from the task. To successfully show that strategy selection can improve effectiveness, the model-based strategy selector must yield a higher score than each of the individual strategies that it chooses from, and to show that the selection model is useful for the strategy selection, the model-based strategy selector must yield a higher score than the random strategy selector.

The random strategy selector Random that we need for Claim 2 chooses randomly with uniform distribution from the set of strategies, ignoring the selection model.

3.2 Benchmark Set

The set of verification tasks that we use in our experiments is taken from the benchmark collection that is also used in SV-COMP. In particular, we use all benchmark categories from SV-COMP 2018 [8] for which we have identified different strategies.

This means that we exclude the category *ConcurrencySafety* as well as the categories for verifying the properties for overflows, memory safety, and termination, for each of which there is only one known suitable strategy in CPAchecker. The remaining set of categories consists of 5 687 verification tasks from the subcategory *DeviceDriversLinux64_ReachSafety* of the category *SoftwareSystems* and from the following subcategories of the category *ReachSafety*: *Arrays*, *Bitvectors*, *ControlFlow*, *ECA*, *Floats*, *Heap*, *Loops*, *ProductLines*, *Recursive*, and *Sequentialized*. A total of 1 501 of these tasks are known to contain a specification violation, and we expect the other 4 186 to satisfy their specification.

3.3 Experimental Setup

For our experiments, we executed version `1.7.6-isola18` of CPAchecker on machines with one 3.4 GHz CPU (Intel Xeon E3-1230 v5) with 8 processing units and 33 GB of RAM each. The operating system was Ubuntu 16.04 (64 bit), using Linux 4.4 and OpenJDK 1.8. We limited each verification run to two CPU cores, a CPU run time of 15 min, and a memory usage of 15 GB. We used the benchmarking framework BenchExec [9] [12] to conduct our experiments, to ensure reliable and accurate measurements.

[8] `https://sv-comp.sosy-lab.org/2018/benchmarks.php`
[9] `https://github.com/sosy-lab/benchexec`

Table 1: Results for all 5 687 verification tasks (1 501 contain a bug, 4 186 are correct), for all basic strategies

Approach	VA-NoCEGAR	VA-CEGAR	PA	KI	BAM	BMC
Score	3 966	5 397	4 881	5 340	1 335	2 484
Correct results	2 365	3 046	2 840	3 053	2 575	1 757
Correct proofs	1 601	2 367	2 073	2 319	2 104	759
Correct alarms	764	679	767	734	471	998
Wrong proofs	0	0	0	0	10	0
Wrong alarms	0	1	2	2	189	2
Timeouts	2 376	1 554	2 497	2 236	2 167	3 379
Out of memory	1	1	14	243	128	381
Other inconclusive	945	1 085	334	153	618	168
Times for correct results						
Total CPU Time (h)	30	54	39	68	33	28
Avg. CPU Time (s)	45	64	49	80	46	57
Total Wall Time (h)	24	44	33	43	25	29
Avg. CPU Time (s)	36	52	42	51	40	51

3.4 Presentation

The full results of our evaluation are available on a supplementary web page.[10] All reported times are rounded to two significant digits. To evaluate the choices of our strategy selector, we use the community-agreed scoring schema of SV-COMP, which assigns quality values to each verification result, i.e., we calculate a score that quantifies the quality of the results for a verification strategy. For every correct safety proof, 2 points are assigned and for every real bug found, 1 point is assigned. A score of 32 points is subtracted for every wrong proof of safety (false negative) and 16 points are subtracted for every wrong alarm (false positive) reported by the strategy, This scoring follows a community consensus [4] on the difficulty of verification versus falsification and the importance of correct results, and is designed to value safety higher than finding bugs, and to punish wrong answers severely.

3.5 Claim 1: Combining Strategies is Effective

In our first experiment we confirm the common knowledge that a sequential combination of several basic strategies can be more effective than either of its components. For this experiment, we compare the verification results of the winning strategy of the 7th Intl. Competition on Software Verification "CPA-Seq", to the results obtained by the basic strategies that it is composed of. Figure 3 shows the quantile functions for these strategies and Table 1 displays the detailed verification results and times for all basic strategies, whereas Table 2 contains the corresponding data for CPA-Seq and other combinations of strategies. We observe that CPA-Seq clearly outperforms the other strategies used in this experiment,

[10] `https://www.sosy-lab.org/research/strategy-selection/`

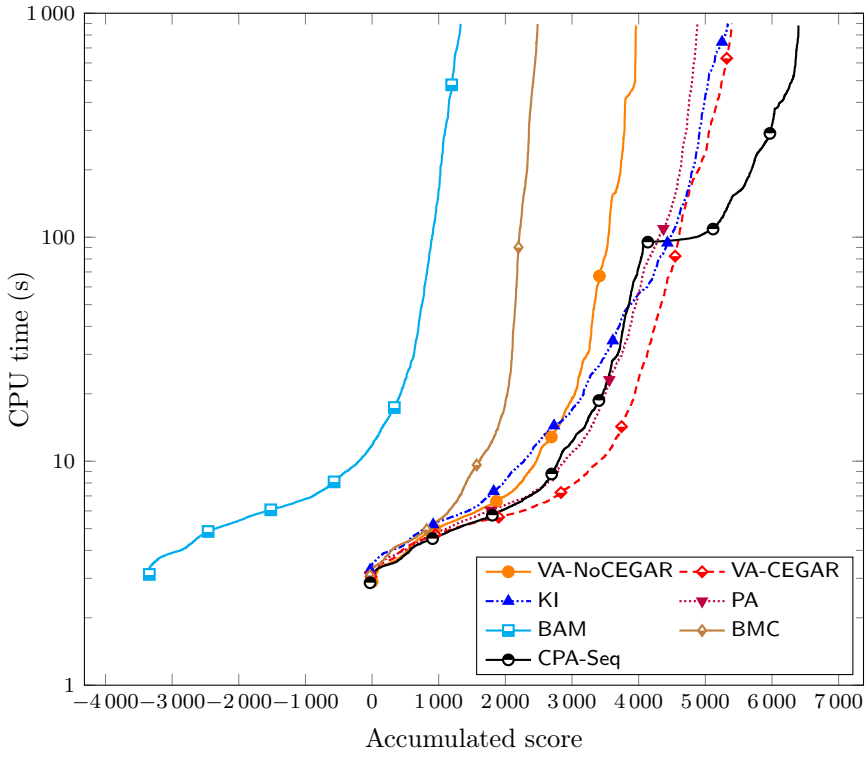


Fig. 3: Quantile functions of different individual strategies and one sequential combination of those strategies (CPA-Seq), as well as one further individual strategy (BMC), for their accumulated scores showing the CPU time for the successful results, offset to the left by the total penalty for incorrect results of each corresponding strategy

even though it is only a sequential combination of the other strategies and contains no added features. We make the same observation for VA-BAM-KI, which is better than each of VA-NoCEGAR, VA-CEGAR, BAM, and KI. While BMC-BAM-PA is better than its main component BMC and its fallback component for recursion, BAM, it has a lower score than its other fallback component PA. The large amount of incorrect results produced by BAM and the resulting low score is caused by the lack of support for pointer-alias handling of this component mentioned in the description of strategies in Sect. 2.2, but while it is obviously unsuitable as a standalone strategy, it does add value as a fallback solution for CPA-Seq.

3.6 Claim 2: Strategy Selection is Effective

In our second experiment, we show that (1) using a strategy selector can be more effective than always choosing the same strategy. This is shown by the

Table 2: Results for all 5 687 verification tasks (1 501 contain a bug, 4 186 are correct), for all combinations of basic strategies: simple sequential combinations, random choice between the sequential combinations, model-based strategy selection, and an imaginary oracle that always selects the best of the three strategies for any given task.

Approach	Sequential Combinations			Random	Model-Based	Oracle
	CPA-Seq	BMC-BAM-PA	VA-BAM-KI			
Score	6 399	2 612	6 442	5 174	6 790	7 036
% of Oracle Score	91	37	92	74	97	100
Correct results	3 740	1 840	3 740	3 122	3 932	4 111
Correct proofs	2 691	804	2 734	2 084	2 922	2 957
Correct alarms	1 049	1 036	1 006	1 038	1 010	1 154
Wrong proofs	0	0	0	0	0	0
Wrong alarms	2	2	2	2	4	2
Timeouts	1 715	3 385	1 879	2 317	1 486	1 347
Out of memory	194	406	26	202	224	185
Other inconclusive	36	54	40	44	41	42
Times for correct results						
Total CPU Time (h)	79	28	87	66	99	96
Avg. CPU Time (s)	76	54	83	76	90	84
Total Wall Time (h)	65	25	70	55	80	79
Avg. CPU Time (s)	63	48	67	63	73	69

model-based strategy selector Model-Based, which achieves a higher score than each of the three strategies that it chooses from (compare column Model-Based with the columns CPA-Seq, BMC-BAM-PA, and VA-BAM-KI). Even the strategy selector Random performs better than one of the strategies that it chooses from (compare column BMC-BAM-PA with column Random).

We also show that (2) using our proposed selection model (consisting of a few simple Boolean features) is effective, because the strategy selector based on that model is more effective than a random choice between the three strategies, and also, for all three available choices, more effective than any constant strategy selector (always choosing the same strategy).

As we can see in Fig. 4, this model-based strategy selection pays off and yields a significantly higher score than each of its competitors. Table 2 shows that while this model-based strategy selection still offers room for improvement because it causes two more wrong alarms than the next-best strategy, this drawback is outweighed by the large amount of correct proofs it produces. This shows that even with a very simple set of Boolean features and a very small set of choices, we can already obtain very promising results. Due to the nature of this approach, adding more features to improve the granularity of the classification and adding more strategy choices to take advantage of the ability to complement this fine-grained classification with a better strategy for each class of tasks, can further improve upon our results. Table 2 also contains the column Oracle that

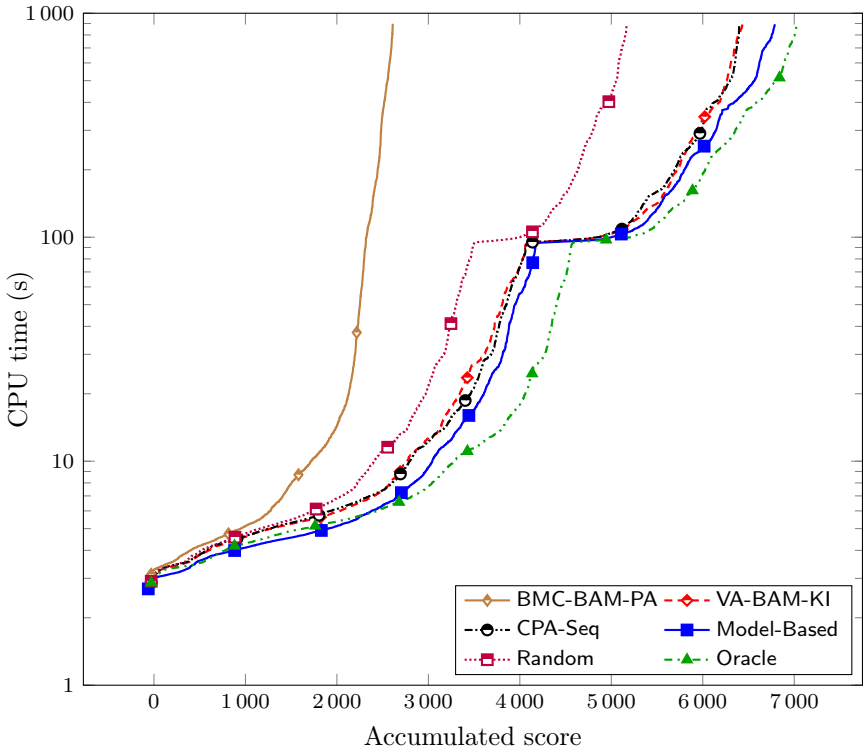


Fig. 4: Quantile functions of three different constant strategy selectors, one model-based strategy selector, one random strategy selector, and one selector based on a hypothetical all-knowing oracle, for their accumulated scores showing the CPU time for the successful results, offset to the left by the total penalty for incorrect results of each corresponding strategy

shows the best results obtainable by an (imaginary) ideal strategy selector based on an oracle that is able to determine the best of the three strategies CPA-Seq, BMC-BAM-PA, and VA-BAM-KI for each task, which achieves only 246 more points than our model-based selector. This means that our model-based selector reaches 97 % of the maximum score achievable by selecting between CPA-Seq, BMC-BAM-PA, and VA-BAM-KI on tasks of our benchmark set.

3.7 Threats to Validity

External Validity. Approaches for strategy selection that are not based on unsupervised learning are dependent on the strategies in the image range that the selector maps to. Therefore, our concrete instantiation of the selector is limited to the chosen strategies and does not consider other strategies of CPAchecker or other software verifiers.

We only showed that our selection model is useful for the given benchmark set. The benchmark set is taken from the largest and most diverse set of verification tasks that is publicly available, but the selection model might not sufficiently well distinguish verification tasks that are different from those in the benchmark set.

Note also that we considered only one verification property in the selection of the benchmark set and in the strategy selector. For benchmark sets with more than one verification property, it may be beneficial to define a strategy selector that considers the verification property as an additional feature to distinguish between tasks.

While the scoring schema from SV-COMP, which we used to model quality, is community agreed and quite stable in its design, a different scoring schema might favor a different strategy-selection function.

Internal Validity. While we used one of the best available benchmarking frameworks, namely BenchExec [11] [12], which is used by several international competitions, to conduct our experiments and ensure reliable and accurate measurements, there still might be measurement errors.

4 Conclusion

This paper explains an approach for strategy selection that is based on a simple selection model —a small set of Boolean features— that is easy to extract statically from the program source code. As strategies to choose from we use the winner of the last competition on software verification (SV-COMP'18) and two more strategies that we constructed from the same verification framework. We evaluated our approach to strategy selection on a benchmark set consisting of 5 687 verification tasks and show that our strategy selector outperforms the winner of the last competition. We hope that this result can be taken as a baseline for comparison of more sophisticated approaches to strategy selection.

References

1. S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. v. Rhein. Domain types: Abstract-domain selection based on variable usage. In *Proc. HVC*, LNCS 8244, pages 262–278. Springer, 2013.
2. T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4% false alarms. In *Proc. FMCAD*, pages 35–42. IEEE, 2010.
3. D. Beyer. Second competition on software verification (Summary of SV-COMP 2013). In *Proc. TACAS*, LNCS 7795, pages 594–609. Springer, 2013.
4. D. Beyer. Software verification with validation of results (Report on SV-COMP 2017). In *Proc. TACAS*, LNCS 10206, pages 331–349. Springer, 2017.
5. D. Beyer, M. Dangl, and P. Wendler. Boosting k-induction with continuously-refined invariants. In *Proc. CAV*, LNCS 9206, pages 622–640. Springer, 2015.

[11] `https://github.com/sosy-lab/benchexec`

6. D. Beyer, M. Dangl, and P. Wendler. A unifying view on SMT-based software verification. *J. Autom. Reasoning*, 60(3):299–335, 2018.
7. D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking: A technique to pass information between verifiers. In *Proc. FSE*, pages 57:1–57:11. ACM, 2012.
8. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Proc. PLDI*, pages 300–309. ACM, 2007.
9. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
10. D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.
11. D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.
12. D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: Requirements and solutions. *Int. J. Softw. Tools Technol. Transfer*, 2017.
13. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.
14. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS*, LNCS 1579, pages 193–207. Springer, 1999.
15. C. Bishop and C. G. Johnson. Assessing roles of variables by program analysis. In *Proc. CSEIT*, pages 131–136. TUCS, 2005.
16. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
17. E. M. Clarke, D. Kröning, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, LNCS 2988, pages 168–176. Springer, 2004.
18. M. Czech, E. Hüllermeier, M. Jakobs, and H. Wehrheim. Predicting rankings of software verification tools. In *Proc. SWAN*, pages 23–26. ACM, 2017.
19. Y. Demyanova, T. Pani, H. Veith, and F. Zuleger. Empirical software metrics for benchmarking of verification tools. In *Proc. CAV*, LNCS 9206, pages 561–579. Springer, 2015.
20. Y. Demyanova, T. Pani, H. Veith, and F. Zuleger. Empirical software metrics for benchmarking of verification tools. *Formal Methods in System Design*, 50(2-3):289–316, 2017.
21. Y. Demyanova, P. Rümmer, and F. Zuleger. Systematic predicate abstraction using variable roles. In *Proc. NFM*, LNCS 10227, pages 265–281, 2017.
22. Y. Demyanova, H. Veith, and F. Zuleger. On the concept of variable roles and its use in software analysis. In *Proc. FMCAD*, pages 226–230. IEEE, 2013.
23. A. Gurfinkel, A. Albarghouthi, S. Chaki, Y. Li, and M. Chechik. Ufo: Verification with interpolants and abstract interpretation (competition contribution). In *Proc. TACAS*, LNCS 7795, pages 637–640. Springer, 2013.
24. B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275(7):51–54, 1997.
25. A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *Proc. CAV*, LNCS 7358, pages 427–443. Springer, 2012.
26. P. Müller, P. Peringer, and T. Vojnar. Predator hunting party (competition contribution). In *Proc. TACAS*, LNCS 9035, pages 443–446. Springer, 2015.
27. A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The yogiproject: Software property checking via static analysis and testing. In *Proc. TACAS*, LNCS 5505, pages 178–181. Springer, 2009.

28. J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.
29. J. Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proc. HCC*, pages 37–39. IEEE, 2002.
30. E. Sherman and M. B. Dwyer. Structurally defined conditional data-flow static analysis. In D. Beyer and M. Huisman, editors, *Proc. TACAS, Part II*, LNCS 10806, pages 249–265. Springer, 2018.
31. T. Stieglmaier. Augmenting predicate analysis with auxiliary invariants. Master's Thesis, University of Passau, Software Systems Lab, 2016.
32. V. Tulsian, A. Kanade, R. Kumar, A. Lal, and A. V. Nori. MUX: Algorithm selection for software model checkers. In *Proc. MSR*. ACM, 2014.
33. A. van Deursen and L. Moonen. Type inference for COBOL systems. In *Proc. WCRE*, pages 220–230. IEEE, 1998.
34. A. van Deursen and L. Moonen. Understanding COBOL systems using inferred types. In *Proc. IWPC*, pages 74–81. IEEE, 1999.
35. P. Wendler. CPAchecker with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In *Proc. TACAS*, LNCS 7795, pages 613–615. Springer, 2013.
36. D. Wonisch and H. Wehrheim. Predicate analysis with block-abstraction memoization. In *Proc. ICFEM*, LNCS 7635, pages 332–347. Springer, 2012.