Construction of Verifier Combinations Based on Off-the-Shelf Verifiers

Dirk Beyer[1] ✉, Sudeep Kanav[1], and Cedric Richter[2]

[1] LMU Munich, Munich, Germany
[2] Carl von Ossietzky University, Oldenburg, Germany

Abstract. Software verifiers have different strengths and weaknesses, depending on properties of the verification task. It is well-known that combinations of verifiers via portfolio and selection approaches can help to combine the strengths. In this paper, we investigate (a) how to easily compose such combinations from *existing*, 'off-the-shelf' verification tools without changing them and (b) how much performance improvement easy combinations can yield, regarding the effectiveness (number of solved problems) and efficiency (consumed resources). First, we contribute a method to systematically and conveniently construct verifier combinations from existing tools, using the composition framework CoVeriTeam. We consider sequential portfolios, parallel portfolios, and algorithm selections. Second, we perform a large experiment on 8 883 verification tasks to show that combinations can improve the verification results *without* additional computational resources. All combinations are constructed from off-the-shelf verifiers, that is, we use them as published. The result of our work suggests that users of verification tools can achieve a significant improvement at a negligible cost (only configure our composition scripts).

Keywords: Software verification · Program analysis · Cooperative verification · Tool Combinations · Portfolio · Algorithm Selection · CoVeriTeam

1 Introduction

Automatic software verification has been an active area of research for many decades and various tools and techniques have been developed to solve the problem of verifying software [3, 7, 9, 25, 34, 37]. The research has also been adopted in practice [2, 22, 24, 39]. Each tool and technique has its own strengths in specific areas. In such a scenario, it becomes obvious to combine these tools to benefit from the strengths of individual tools, leading to a 'meta verifier' that solves more problems. Most current combination approaches are hardcoded, that is, the choice of the tools and the way to combine them is specifically programmed.

We contribute a method to construct combinations in a systematic way, independently from the set of tools to use. As for the types of combinations, we considered sequential and parallel portfolio [36], and algorithm selection [47]. The combinations are composed and executed with the tool CoVeriTeam [15].[1]

[1] https://gitlab.com/sosy-lab/software/coveriteam/

© The Author(s) 2022
E. B. Johnsen and M. Wimmer (Eds.): FASE 2022, LNCS 13241, pp. 49–70, 2022.
https://doi.org/10.1007/978-3-030-99429-7_3

CoVeriTeam is a tool that is based on off-the-shelf atomic actors, which are executable units based on tool archives. It provides a simple language to construct tool combinations, and manages the download and execution of the existing tools on the provided input. CoVeriTeam provides a library of atomic actors for many well-known and publicly available verification tools. A new verification tool can be easily integrated into CoVeriTeam within a few minutes of effort.

For our experimental evaluation, we selected eight of the verification tools that participated in the 10th competition on software verification [6]. We reused the archives submitted to this competition, and composed combinations of three types (sequential and parallel portfolio, algorithm selection) with 2, 3, 4, and 8 verification tools: in total 12 combinations. We evaluated these 12 combinations on a large benchmark set consisting of 8 883 verification tasks in total and compared the results of the combinations against the results of the existing tools.

We show that all three combination approaches can lead to considerable improvements of the performance regarding effectiveness (number of correctly solved instances) and efficiency (consumed resources).

Contributions. We make the following contributions:

1. We show how to conveniently construct combination approaches from off-the-shelf verification tools in a modular manner, without changing the tools.
2. We perform an extensive comparative evaluation of sequential portfolio, parallel portfolio, and algorithm selection approaches.
3. A reproduction package containing the tools and experiment data.

2 Improving Verification by Verifier Combinations

In this study, we explore different strategies for combining verifiers to improve the overall verification effectiveness. We focus on the most commonly applied *black-box* combinations (i.e., combinations that do neither require any changes to the existing tools nor communication between verification tools) which we briefly describe in the following.

Verifier Combinations. Existing strategies for combining verifiers can be generally classified into one of the following three categories: *sequential portfolios* [17, 33, 53], *parallel portfolios* [35, 36, 40], and *algorithm selectors* [8, 28, 47, 48, 50]. We provide an overview over these composition strategies in Figs. 1 and 2.

Sequential Portfolio. Portfolios combine several verification algorithms by executing them either sequentially or in parallel. A sequential portfolio (Fig. 1) executes a set of verifiers sequentially by running one verifier after another. In this setting, each verifier is assigned a specific time limit and the verifier runs until it finds a solution or reaches the time limit. If the current verifier is able to solve the given verification task, the sequential composition is stopped and the solution is emitted. Otherwise, if a verifier runs into a timeout without, the current algorithm is stopped and the next one is started. CPA-Seq [17, 53] and Ultimate Automizer [33] are examples of sequential portfolios.

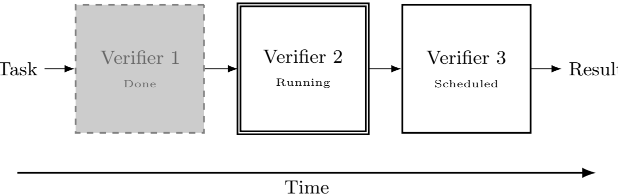


Fig. 1: Sequential portfolio of verifiers. Each verifier runs for a certain amount of time. If a verifier stops without computing a result (grey box), the next one is started (white box with double borders).

Parallel Portfolio. In contrast to sequential portfolios, a parallel portfolio (Fig. 2(a)) executes all verification algorithms in parallel, while sharing all system resources like CPU time and memory. As soon as one algorithm solves the given verification problem, the portfolio is stopped. Based on the assumption that all verifiers provide only sound solutions, we can safely take the first solution computed as the final result of the overall portfolio. PredatorHP [35, 40] is an example of a parallel portfolio.

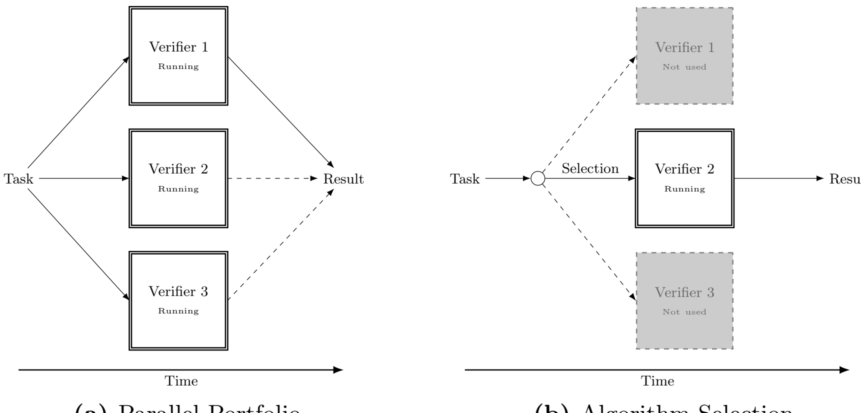
Algorithm Selection. To reduce spending resources on unsuccessful verifiers, algorithm selectors (Fig. 2(b)) are designed to select the verification algorithm that is likely well suited to solve a given verification task. More precisely, the algorithm selector analyzes the given verification problem for common characteristics (typically program features like the existence of a loop or an array) and based on these features, selects a verification algorithm likely suited for the given problem. Then the selected verifier is executed. Algorithm selectors were recently explored for selecting a task-dependent verification algorithm (e.g., in PeSCo [48, 50]) or a complete verification strategy (e.g., in CPAchecker [8]).

The above combination types have their own advantages and limitations when applied in real-world scenarios. While algorithm selectors omit the necessity of sharing resources, the approach heavily relies on the used selection algorithm. If the selection algorithm is not powerful enough or the selection task is too difficult, the selector fails to identify a verifier equipped for the given task. Although portfolios omit this problem by assigning the verification task to several verifiers, each verifier gets less resources, which could lead to out-of-resource failures.

3 Construction of Verifier Combinations with CoVeriTeam

CoVeriTeam [15] is a tool for creating and executing tool combinations for cooperative verification [20]. It consists of a language for tool composition, and an execution engine for this language. Tools are considered as verification actors (verifiers, validators, testers, transformers), and the inputs consumed and outputs produced by the tools as verification artifacts (programs, specifications, witnesses, results). Verification artifacts are seen as basic objects, verification actors as basic operations, and tool combinations as composition of these operations.

CoVeriTeam supports execution of most of the well known automated verification tools that are publicly available. The composition operators supported by



(a) Parallel Portfolio **(b)** Algorithm Selection

Fig. 2: Comparison of parallel portfolio and algorithm selection

CoVeriTeam are: SEQUENCE, PARALLEL, REPEAT, and ITE. SEQUENCE executes the composed tools sequentially, PARALLEL in parallel, REPEAT repeatedly till a termination condition is satisfied, ITE is an *if-then-else* that executes one tool if the provided condition is true and otherwise the other. The work in this paper uses SEQUENCE, PARALLEL, ITE, and a newly developed PORTFOLIO.

3.1 Verifier Based on Sequential Portfolio

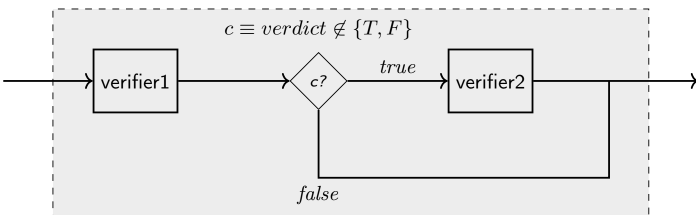


Fig. 3: Verifier based on sequential portfolio

Figure 3 shows the construction of a sequential portfolio of two verifiers verifier1 and verifier2 using CoVeriTeam. This construction uses two kinds of compositions: SEQUENCE and ITE. At the outermost level, it is a sequence of verifier1 and an actor that in itself is a composition—an ITE composition. Let us call it ite_verifier. When we execute this composition, first, verifier1 is executed and then ite_verifier. ite_verifier first checks if verifier1 was successful in verification or not (i.e., $verdict \notin \{T, F\}$). If verifier1 was successful, then it forwards the results, otherwise, verifier2 is executed and its results are taken. This construction can be generalized to create sequential portfolios of arbitrary sizes. We used it to create sequential portfolios of 2, 3, 4, and 8 verifiers.

3.2 Verifier Based on Parallel Portfolios

We developed a composition operator for parallel portfolio in CoVeriTeam. In this composition, multiple tools are executed in parallel and the result of the

one that succeeds first is taken. The composition consists of a set of verification actors of the same type (verifiers, testers, etc.), and a success condition defined over the artifacts produced by these actors. When one actor finishes, the success condition is evaluated: if it holds then the output of this actor is taken and the execution of the remaining actors is stopped. Otherwise, the portfolio waits for the next actor to finish and repeats the check. If none of the actors produce the output that satisfies the success condition, the result of the last one is taken.

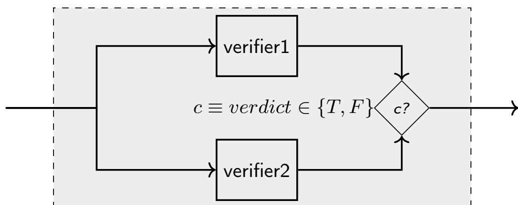


Fig. 4: Verifier based on parallel portfolio

Figure 4 shows a parallel portfolio of two verifiers verifier1 and verifier2. In this case, both the verifiers are executed simultaneously. When one verifier finishes, its result is checked for the success condition (i.e., $verdict \in \{T, F\}$). If the success condition holds then the result is forwarded, otherwise, the result is discarded and we wait for the second verifier to finish. Once a successful result is available, the remaining executing verifiers are terminated. For our experiments, we created parallel portfolios of 2, 3, 4, and 8 tools.

3.3 Verifier Based on Algorithm Selection

We designed and implemented a generic selection framework in CoVeriTeam for selecting verifiers. The framework decomposes the algorithm-selection process into two phases: (1) a *feature-extraction phase*, in which a feature encoder extracts a set of predefined features for a given verification task (i.e., certain characteristics that are believed to indicate difficulty for a verifier), and (2) *selection* to identify an appropriate verifier based on the extracted features. Each phase is constructed using CoVeriTeam actors (explained below in more detail). Figure 5 shows the CoVeriTeam composition of a verifier based on algorithm selection.

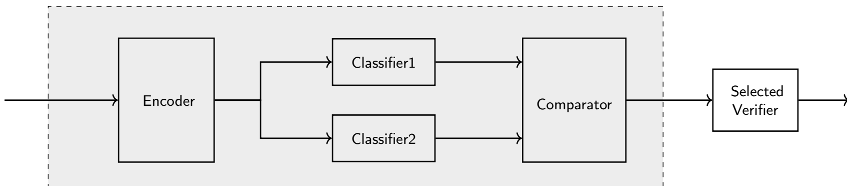


Fig. 5: Verifier based on algorithm selection

Feature Encoder. The first component of our framework is the feature *encoder*. Given a verification task consisting of a program P and a specification S, the goal of the feature encoder is to encode the problem into a meaningful feature-vector

(FV) representation, which we can later use to select a verification tool. Typically, the representation encodes certain features of a program which might correlate with the performance of a verifier such as the occurrence of specific loop patterns [28] or variable types [29]. In this study, we encode verification problems via a learning-based feature encoder by employing a pretrained *CSTTransformer* [50]. The CSTTransformer first parses a given program P into a simplified abstract syntax tree (AST) representation. Afterwards, a specific type of neural network processes the AST structure to produce a vector representation. The last encoding step is learned by pretraining the neural network on selecting various verification *tools*. While this approach was originally developed to learn a vector representation optimized for a specific verifier composition, the authors showed that the learned encoder can be effectively reused across many new selection tasks, often outperforming other hand-crafted feature encoders.

Selection of Verifiers Based on the Individual Difficulty of the Tasks. The same task might be solved with one tool in a few seconds, while another is not able to find a solution within the given resource constraints. Therefore, to avoid wasting resources on tools that are not well suited for a given task, the algorithm selector aims to predict the difficulty of a task before executing a tool. Then, the tool that is predicted to be the best suited tool for the task is executed.

Similar to previous work [28, 50], we learn to predict the difficulty of task with *hardness models* [55]. Based on the previously computed vector representation, a hardness model learns to predict the hardness of a given task for a specific tool. In our case, this reduces to a binary classification problem of predicting whether a tool can solve a task or not. We address this by training logistic regression classifiers. The classifier's confidence that a verifier will fail a particular task then determines the hardness of the task.

Now, given a set of hardness models —each accessing the hardness of a verification task for a specific tool— a verification tool is selected for which the task is likely easy (i.e., the respective model outputs the lowest hardness score). The final selection is done by a comparator implemented in CoVeriTeam that selects a tool by comparing the hardness scores.

3.4 Extensibility

To facilitate future research and the design of novel combinations, we implemented all combination types such that they can be easily configured and extended. Extending a combination with a new verifier only requires an actor definition for that verifier in CoVeriTeam. Afterwards, this actor can be put in a sequential or parallel portfolio by adding it to the composition. While our algorithm selector can be easily used with all tools employed during our experiments, extending a combination based on algorithm selection with a new verifier requires a bit more effort. However, by using hardness models together with a common feature representation we simplified the process required for configuring algorithm selection. In fact, we are able to modify the set of verifiers to select from by simply adding or removing individual hardness models. While previous approaches to

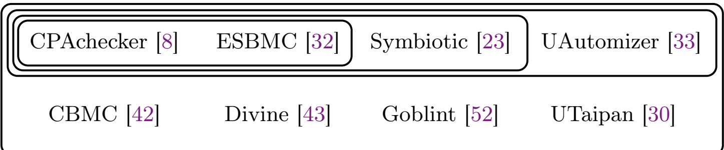


Fig. 6: Subsets of verification tools used for composition

verifier selection often require training the complete selector from scratch, our combination can be extended by training a single hardness model.[2] For training a new model, we provide all training scripts that were used for training our hardness models and a precomputed dataset of vector representations for SV-COMP 2021. Therefore, to integrate a new tool in our algorithm selector, one only requires to run the respective verifier once on (a subset of) the benchmark set. The results then act as training examples.

4 Evaluation

We perform a thorough experimental evaluation on a large benchmark set in order to show the potential of combinations. We address the following research questions concerning the comparative evaluation of combinations against standalone tools:

RQ 1. Can a CoVeriTeam-based sequential portfolio of verifiers perform significantly better than standalone tools with respect to
- (a) number of solved verification tasks, and
- (b) resource consumption?

RQ 2. Can a CoVeriTeam-based parallel portfolio of verifiers perform significantly better than standalone tools with respect to
- (a) number of solved verification tasks, and
- (b) resource consumption?

RQ 3. Can a CoVeriTeam-based algorithm selection of verifiers perform significantly better than standalone tools with respect to
- (a) number of solved verification tasks, and
- (b) resource consumption?

4.1 Experimental Setup

Selection of Existing Verifiers. We selected eight existing verification tools that performed well in a recent competition on software verification (SV-COMP 2021) [6]. We excluded two verifiers from consideration: VeriAbs [27] and PeSCo [49]. VeriAbs was excluded because its license does not allow us to use it for scientific evaluation, and PeSCo because it is a derivate of CPAchecker that would not contribute to diversity of technology in the combinations. The chosen set of verifiers used for the tool combinations is depicted in Fig. 6.

[2] A single hardness model can be trained within a few minutes on a modern CPU.

Tool Combinations. We evaluated twelve verifier combinations: for each of sequential portfolio, parallel portfolio, and algorithm selection, we constructed a combination of 2, 3, 4, and 8 verifiers. These variants of combinations with different numbers of verifiers allowed us to quantify the influence of the number of verifiers on the performance. We constructed these subsets of verifiers to maximize the number of tasks (from our benchmark set) that can be solved by at least one tool in the subset. For sequential portfolios, we additionally rank the verifiers in descending order of their success on the benchmark. We used the results from SV-COMP 2021 to achieve this. Figure 6 illustrates the sets of verifiers that we composed in different types of combinations.

Execution Environment. Our experiments were executed on machines with the following configuration: one 3.4 GHz CPU (IntelXeon E3-1230 v5) with 8 processing units (virtual cores), 33 GB RAM, operating system Ubuntu 20.04. Each verification run (execution of one tool or combination on one verification task) was limited to 8 processing units, 15 min of CPU time, and 15 GB memory. This configuration is the same as the configuration used in SV-COMP 2021 allowing us to use the competition results of the standalone tools for comparison.

Benchmark Selection. Our benchmark set consists of all the verification tasks with specification `unreach-call` from the open-source collection of verification tasks SV-Benchmarks[3]. Each verification task consists of a program written in C and a specification. The specification is a safety property describing that an error location should never be reached. The benchmark set includes all verification tasks of the competition categories *ReachSafety* and *Concurrency*, and a part of the verification tasks in category *SoftwareSystems*. In total, there were 8 883 verification tasks in our benchmark set. We evaluated our combinations on the version of the benchmark set that was used in SV-COMP 2021 (tag `svcomp21`).

Scoring Schema. We not only count the number of results of each kind[4] for the verification tasks, but also the scores as used in the competition, because this models what the community considers as quality. A verifier is rewarded score points as follows: 2 score points for each correct proof, 1 score point for each correct alarm, -32 score points for wrong proofs, and -16 score points for wrong alarms. This schema has been used in SV-COMP [6] since a few years and has been accepted by the verification community for judging the quality of results.

Resource Measurement and Benchmark Execution. We used the state-of-the-art benchmarking framework BenchExec [18] for executing our benchmarks. It executes tools in isolation, reports the resource consumption, and also enforces the resource limitations. It provides measurements of the consumption of CPU time, wall time, memory, and CPU energy during an execution of a tool.

4.2 Results of Existing Verifiers as Standalone

Table 1 shows the summary of results of the execution of the standalone tools on the selected benchmark set. These results are publicly available in the respective

[3] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

[4] Either claims of program correctness or alarms of specification violations.

Table 1: Standalone verifiers

Verifier	CPAchecker	Esbmc	Symbiotic	UAutomizer	Cbmc	Divine	Goblint	UTaipan
Score	**9 040**	6 623	4 878	7 146	4 663	3 679	2 770	5 338
Correct results	**5 652**	4 481	3 001	4 358	3 484	2 922	1 385	3 725
Correct proofs	**3 516**	2 958	1 909	2 836	1 499	1 605	1 385	2 365
Correct alarms	2 136	1 523	1 092	1 522	1 985	1 317	0	1 360
Wrong results	8	29	2	2	19	41	**0**	24
Wrong proofs	**0**	22	**0**	1	1	12	**0**	23
Wrong alarms	8	7	2	**1**	18	29	0	**1**
Total resource consumption for correct results								
CPU time (h)	190	57	22	97	31	60	**11**	81
Wall time (h)	140	57	22	59	31	15	**11**	52
Memory (GB)	7 000	1 800	770	4 300	1 300	2 000	**120**	2 700
CPU Energy (KJ)	7 700	2 500	1 000	3 500	1 300	1 500	**560**	3 000
Median resource consumption for correct results								
CPU time (s)	61	0.84	0.81	36	**0.70**	17	0.78	39
Wall time (s)	32	0.84	0.84	12	**0.69**	9.1	0.80	13
Memory (MB)	600	53	**25**	450	44	670	**25**	430
CPU Energy (J)	590	11	11	310	**9.2**	150	11	330
Resource consumption of correct results per score point								
CPU time (s/sp)	77	31	16	49	24	59	**15**	55
Wall time (s/sp)	55	31	16	30	24	**14**	15	35
Memory (MB/sp)	780	270	160	600	280	540	**42**	500
Energy (J/sp)	850	380	210	490	280	420	**200**	560

reproduction package of the competition [5] and on the competition web site[5]. We only adjust the presentation to our needs here.

Figure 7 shows the quantile plots of the results, where the x-coordinate represents the quantile of score obtained by the tool below the run time represented by y-coordinate. We used a logarithmic scale for time ranges between 1 and 1000 seconds, and linear scale between 0 and 1 second. The graph of a tool that solves more verification tasks will be farther to the right, and the plot of the faster tools would be lower. The farther on the right side a plot goes and the lower a plot remains. the better it is. More details about these plots are given elsewhere [4].

Figure 8 shows the resource consumption for standalone tools using a parallel-coordinates plot (each parallel coordinate represents a different variable). The plot shows the number of unsolved tasks, and resource consumption per score point. The lower the plot of a tool is the better it is for the user.

4.3 RQ 1: Evaluation of Sequential-Portfolio Verifier

We now present the results of the sequential-portfolio verifier against the existing standalone verifier with the highest score: CPAchecker.

[5] https://sv-comp.sosy-lab.org/2021/results/results-verified

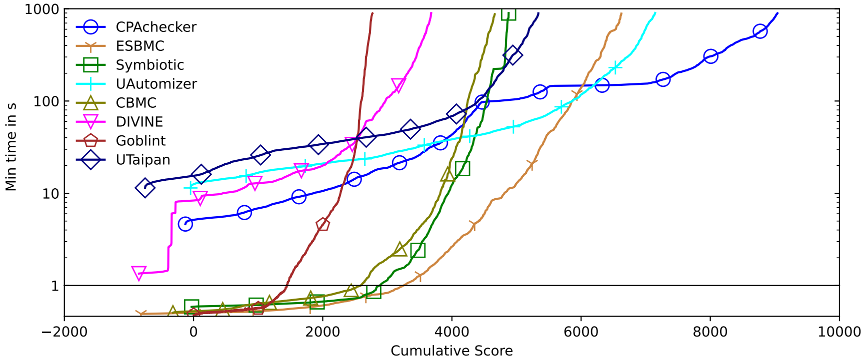


Fig. 7: Standalone verifiers: Score-based quantile plot for results

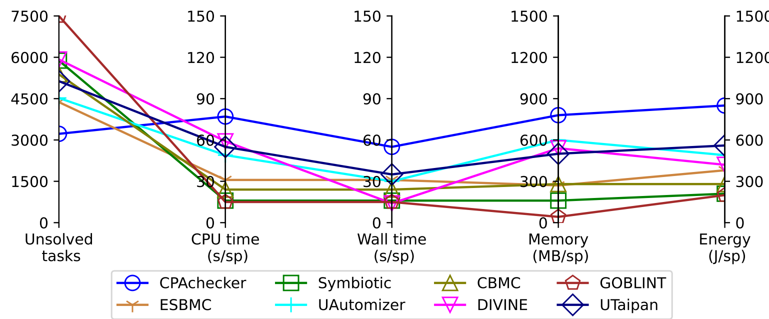


Fig. 8: Standalone verifiers: Parallel-coordinates plot showing unsolved tasks and resource consumption per score point

Table 2 shows the summary of results for the sequential verifiers. The sequential portfolio, in general achieves better score than the best performing standalone tool. The portfolio with 8 tools performs worst, which is expected because as we increase the size of the portfolio, the amount of time allocated to each verifier also decreases. This means that the verifiers can only solve relatively easier tasks. The table also shows that the portfolio requires more resources to solve the tasks. This is a side effect of the sequential portfolio, as all the resources consumed by unsuccessful attempts to solve a given task by the verifiers in a sequence are still counted in the resource consumption. Also, the portfolio with 8 tools has a considerably large number of wrong results as it is reduced to fast results, instead of the verifier earlier in the sequence. The index at which a verifier is placed plays a key role in the performance of the sequential portfolio. If we put a verifier that produces results fast but has more wrong results first in the sequential portfolio, then the overall results are going to have a lot of wrong results.

Figure 9 shows the quantile plot of scores. As a portfolio is biased towards the verifiers that compute results fast and not towards correctness, we see the sequential portfolio combinations starting from farthest in the left, i.e., having the most negative score, or most wrong results. CPAchecker has the least number of wrong results, and because of it its starting point is farthest to the right.

Table 2: Sequential portfolios of different sizes with CPAchecker

Verifier	CPAchecker	Sequential Portfolio of			
		2	3	4	8
Score	9 040	9 198	9 519	**9 522**	8 349
Correct results	5 652	6 058	6 239	**6 275**	6 084
Correct proofs	3 516	3 780	**3 920**	3 903	3 721
Correct alarms	2 136	2 278	2 319	**2 372**	2 363
Wrong results	**8**	26	26	27	61
Wrong proofs	**0**	14	14	14	30
Wrong alarms	**8**	12	12	13	31
Total resource consumption for correct results					
CPU time(h)	**190**	240	260	240	190
Wall time (h)	**140**	190	210	190	150
Memory (GB)	**7 000**	8 900	8 600	8 500	7 600
CPU Energy (KJ)	**7 700**	9 700	11 000	10 000	7 900
Median resource consumption for correct results					
CPU time(s)	**61**	95	100	100	97
Wall time (s)	**32**	54	69	70	54
Memory (MB)	**600**	920	930	910	840
CPU Energy (J)	**590**	920	1 100	1 100	920
Resource consumption of correct results per score point					
CPU time (s/sp)	**77**	95	97	90	82
Wall time (s/sp)	**55**	72	78	72	64
Memory (MB/sp)	**780**	970	910	890	920
CPU Energy (J/sp)	**850**	1 100	1 100	1 100	950

Figure 10 shows that CPAchecker is more resource efficient in comparison to the sequential portfolio. The sequential combination with best score is performing worst in resource efficiency.

4.4 RQ 2: Evaluation of Parallel-Portfolio Verifier

We now present the results of the parallel-portfolio verifiers. The parallel portfolio, mostly, achieves worse score than the best performing standalone tool. But the parallel portfolio with 3 tools scores better. The parallel portfolio is affected by two aspects: (1) size of the parallel portfolio — if too many tools are used then any of them would not get enough resources to verify the task, (2) selection of tools — if there is a fast tool that produces a lot of wrong results it reduces the score. Parallel portfolio, in general, produces more wrong results; even more than sequential portfolio, as the tools are running in parallel, whereas in sequential portfolio this can be somewhat mitigated by putting a more sound tool before a less sound tool. Table 3 shows the summary of results for the parallel portfolios.

Figure 11 shows that parallel portfolios have many more wrong results when compared to CPAchecker. Interestingly, the graph for *ParPortfolio-3*, the best performing parallel portfolio, remains lower than CPAchecker, i.e., it takes less

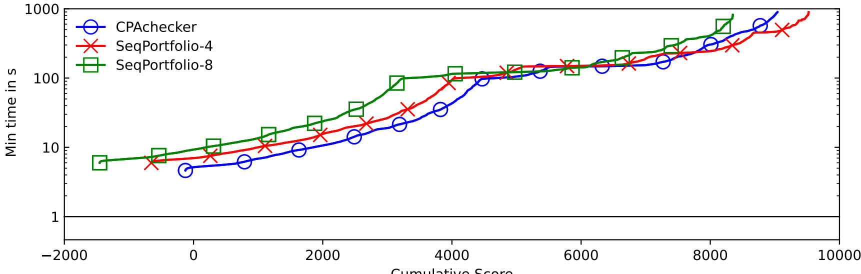


Fig. 9: Sequential portfolios: Score-based quantile plot comparing the best and the worst sequential portfolio (SeqPortfolio-4 and SeqPortfolio-8, respectively) with the best performing standalone tool (CPAchecker)

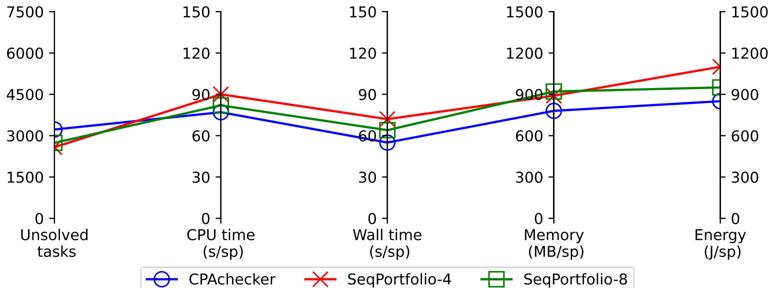


Fig. 10: Sequential portfolios: Parallel-coordinates plot showing unsolved tasks and resource consumption per score point for best and worst portfolio (SeqPortfolio-4 and SeqPortfolio-8, resp.) and the best standalone tool (CPAchecker)

CPU time. This is because the parallel portfolio takes results of the most efficient tool. Figure 12 shows that the best performing parallel portfolio performs better than CPAchecker in terms of resource efficiency except memory consumption.

4.5 RQ 3: Evaluation of Algorithm Selection Verifier

We now present the results of the algorithm-selection verifier. Table 4 shows the summary of results for algorithm selection: There is a clear trend of better results with more verifiers. This is expected because our selector that was trained using machine learning has more options to choose from, and can choose the better one. Also, an algorithm-selection verifier does not need to share resources for the verification task. It needs to perform the prediction, which takes some resources; but after this step all the provided resources are available to the verifier. The number of wrong results is also comparable with CPAchecker, as the training process is biased towards selecting the verifiers that are correct.

In Fig. 13, all the plots start from around similar scores but at different times. Initially, CPAchecker performs better with respect to CPU time, but after around half the scores, algorithm selection starts being more efficient. Figure 14 shows that algorithm selection is also more resource efficient than CPAchecker.

Table 3: Parallel portfolios of different size with CPAchecker

Verifier	CPAchecker	Parallel Portfolio of			
		2	3	4	8
Score	9 040	8 969	**9 459**	8 952	7 547
Correct results	5 652	6 101	**6 363**	6 001	5 367
Correct proofs	3 516	3 780	**3 992**	3 639	3 236
Correct alarms	2 136	2 321	**2 371**	2 362	2 131
Wrong results	**8**	36	35	28	42
Wrong proofs	**0**	21	21	15	24
Wrong alarms	**8**	15	14	13	18
Total resource consumption for correct results					
CPU time(h)	190	**160**	170	250	280
Wall time (h)	140	74	**61**	74	64
Memory (GB)	**7 000**	8 900	11 000	14 000	11 000
CPU Energy (KJ)	7 700	5 400	**5 200**	6 500	6 400
Median resource consumption for correct results					
CPU time(s)	61	18	**16**	70	130
Wall time (s)	32	5.2	**4.6**	16	23
Memory (MB)	600	430	**420**	1 000	1 300
CPU Energy (J)	590	140	**120**	470	780
Resource consumption of correct results per score point					
CPU time (s/sp)	77	**65**	66	99	130
Wall time (s/sp)	55	30	**23**	30	31
Memory (MB/sp)	780	**1 000**	1 200	1 500	1 400
CPU Energy (J/sp)	850	600	**550**	720	850

4.6 Discussion

The experiments show that each of the compositions has a configuration that can perform better than any standalone tool in terms of correctly solved tasks. Initially, we thought that portfolios would be less resource efficient than standalone tools, and, in particular, would not be able to solve hard tasks as the resources allocated to each tool would be less. But the experimental data support the opposite: The benchmark set had a few such tasks: for most of the tasks that were hard for one tool, there was some other tool that solved it in the given time. This was especially pronounced in the parallel portfolio. The verifiers in the portfolios have to be selected with different strengths, otherwise there is no benefit, it might even perform worse.

Both the portfolios prefer fast results, as there is no selector. To mitigate this, one needs to either select the tools carefully or add a validation step.

Our algorithm selection was based on a model trained using machine learning. The training penalized the tools that produced more incorrect results, but it did not consider the resource consumption of these tools. In comparison to both the portfolios, the verifier based on algorithm selection produced much less incorrect

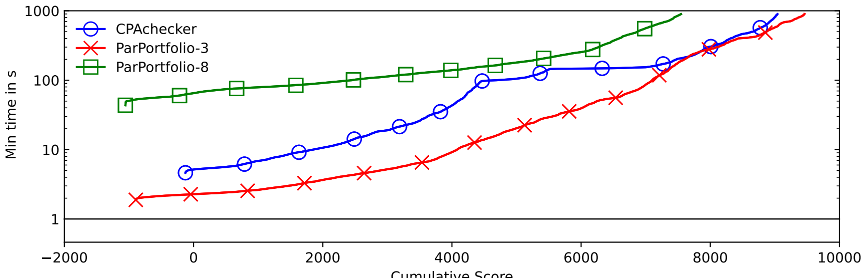


Fig. 11: Parallel portfolios: Score-based quantile plot comparing the best and the worst performing parallel portfolios (ParPortfolio-3 and ParPortfolio-8, respectively) with the best performing standalone tool (CPAchecker)

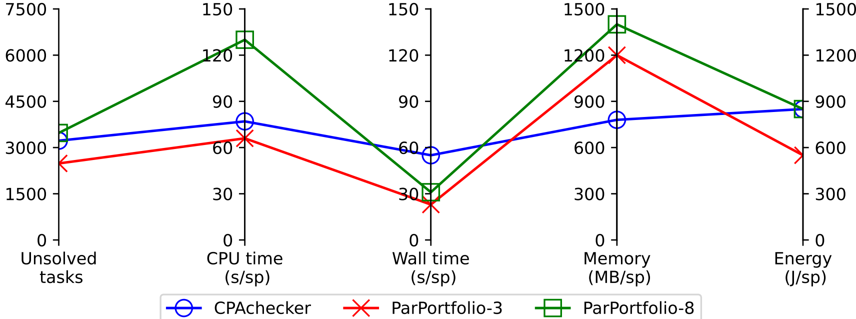


Fig. 12: Parallel portfolios: Parallel-coordinates plot showing unsolved tasks and resource consumption per score point of best and worst portfolio (ParPortfolio-3 and ParPortfolio-8, resp.) and the best standalone tool (CPAchecker)

results. We think if we used the resource consumption data in our training, the verifier based on selection would have consumed less resources. Our verifier combinations are easy to construct by simply selecting tools that complement each other well. Although this strategy is simple, we found that it still leads to successful combinations for all evaluated combination types. Nevertheless, the combinations can be further fine-tuned to achieve even better results.

The portfolio compositions are easy to construct, and with a well diversified tool selection, portfolios can perform good. Also, the portfolios should not be too large unless we are willing to increase the resources. On the other hand, training the selection requires more preliminary work but with limited resources and enough choice (number of tools) the selection-based verifier works better.

5 Threats to Validity

External Validity. A combination of tools can only be as good as the parts it is combined from. Therefore, the concrete instantiation of our tool combinations is limited by the selected tools and their configuration. We have selected eight of the

Table 4: Algorithm-selection-based verifiers of different sizes with CPAchecker

Verifier	CPAchecker	Algorithm Selection of			
		2	3	4	8
Score	9 040	9 226	9 689	9 816	**9 886**
Correct results	5 652	5 904	6 086	6 125	**6 214**
Correct proofs	3 516	3 658	3 843	3 867	**3 896**
Correct alarms	2 136	2 246	2 243	2 258	**2 318**
Wrong results	**8**	15	11	8	11
Wrong proofs	**0**	6	4	3	3
Wrong alarms	**8**	9	7	5	**8**
Total resource consumption for correct results					
CPU time(h)	**190**	200	200	200	210
Wall time (h)	**140**	160	160	150	170
Memory (GB)	7 000	6 900	6 900	6 200	**6 000**
CPU Energy (KJ)	**7 700**	8 200	8 600	8 400	9 000
Median resource consumption for correct results					
CPU time(s)	61	**47**	48	66	55
Wall time (s)	32	**30**	**30**	35	42
Memory (MB)	600	740	700	550	**420**
CPU Energy (J)	**590**	490	500	660	620
Resource consumption of correct results per score point					
CPU time (s/sp)	77	77	76	**73**	76
Wall time (s/sp)	**55**	61	61	56	63
Memory (MB/sp)	780	750	720	630	**600**
CPU Energy (J/sp)	**850**	890	890	**850**	910

most powerful verification tools as determined by the annual software-verification competition, and executed them in the original configuration as submitted to the competition. Furthermore, our evaluation results only hold for the given benchmark set. While we have evaluated our tool combinations on programs taken from one of the largest and diverse verification benchmarks publicly available, the performance of the evaluated combinations might differ on other sets of tasks.

Similarly, this also impacts the training of our algorithm selector. The training of a learning-based algorithm selector, which we employ for tool combinations based on algorithm selection, requires a large and diverse set of verification tasks; and each task has to be labeled with the execution results of each tool in our combination. The used benchmarks repository[6] was created by the efforts of the verification community over many years. We are not aware of any other benchmark set of verification tasks that is as diverse as this one. As a result, we had to train our algorithm selector on the same dataset that we later use for benchmarking the tool combinations. Therefore, we only showed that algorithm selection improves the performance of verification on the given benchmark set

[6] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

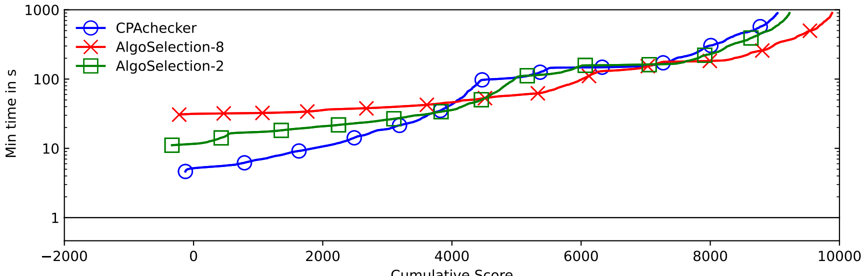


Fig. 13: Algorithm-selection-based verifiers: Score-based quantile plot comparing the best and the worst performing portfolio (AlgoSelection-8 and AlgoSelection-3, respectively) with the best performing standalone tool (CPAchecker)

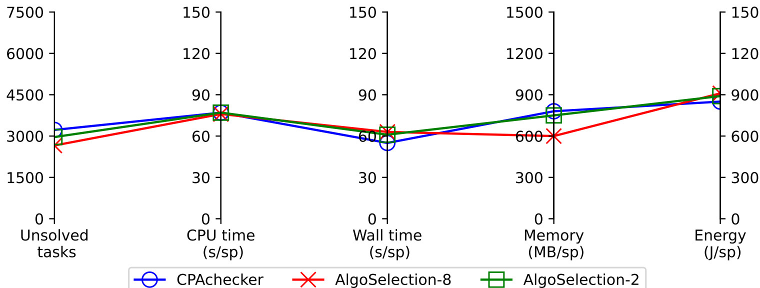


Fig. 14: Algorithm-selection-based verifiers: Parallel-coordinates plot showing unsolved tasks and resource consumption per score point of the best and the worst performing algorithm selection (AlgoSelection-8 and AlgoSelection-2, respectively) and the best performing standalone tool (CPAchecker)

and the selector might only generalize to a set of tasks with similarly distributed verification tasks. For a fair comparison, we (1) restricted the training to linear models, which are known to generalize well, (2) train only on a random subset of the benchmark, and (3) cross validated our model over multiple benchmark splits. The variance of selection performance between different splits was less than 1 %. Therefore, the performance of our trained algorithm selector is likely independent of the random subset selected for training.

Finally, the evaluation of algorithm selection is dependent on the chosen selection methodology and choosing alternative selection methods, for example, based on hand-crafted rules, might impact the evaluation. However, the design of hand-crafted methods is not straightforward and might require deep expert knowledge about the tool implementation. Depending on the human designer, this design process might in addition be biased in favor of certain tool combinations, which could also impact the experimental results.

For sequential portfolios, we ordered verifiers in sequence according to their performance in SV-COMP 2021. Changing the order of the tools might change the results with respect to resource consumption as well as soundness.

Internal Validity. We have used the same verifier archives, benchmark set, benchmarking framework, resource limits, and infrastructure to execute our experiments as was used in SV-COMP 2021. This minimizes the influence of a changing environment on our experiments, allowing us to compare results of our verifier combinations to the results of the standalone tools from SV-COMP 2021.

CoVeriTeam induces an overhead of about 0.8 s for each actor in the composition, and around 44 MB memory overhead [15]. It is possible that one can reduce this overhead by using shell scripts, but we decided in favor of using CoVeriTeam for composing tools because of the modular design. This is especially pronounced in our algorithm-selector composition. We could have saved a few seconds if we were using a monolithic algorithm selector instead of composing one.

6 Related Work

Combination Strategies for Software Verification. Combining verifiers to increase the verification performance is well established in the domain of software verification [1, 8, 20, 26, 31, 33, 46, 48, 49, 53]. In fact, the top three winning entries of the software-verification competition SV-COMP 2021 all combine various verification techniques to achieve their performance [6]. CPAchecker [8] combines up to six different verification appraoches into three sequential portfolios that are task-dependently selected with an algorithm selector. PeSCo [49] ranks verification algorithms according to their predicted likelihood of solving a given task and then executes them sequentially in descending order. Ultimate Automizer [33] employs an integrated tool chain of preprocessing and verification algorithm to solve a given task. PredatorHP [46] and UFO [1] demonstrate that parallel portfolios can also be a promising strategy when running multiple specialized algorithms at the same time. Even though previous work showed that internal combinations can be successfully applied to improve the effectiveness of a single tool, we show that similar combinations can be effectively employed to combine 'off-the-shelf' verifiers. This gives us the unique opportunity to further increase the number of verifiable programs by simply combining state-of-the-art verification tools.

Cooperative methods [20] distribute the workload of a single verification task among multiple algorithms to combine their strengths. For example, conditional model checking [11, 12, 13, 14] runs two or more verifiers in sequence, while the program is reduced after every step to the state space of program unexplored by the previous algorithm. CoVeriTest [10], a tool for test-case generation based on verification, interleaves multiple verifiers, while (partially) sharing the analysis state between algorithms. MetaVal [19] integrates verification tools for witness validation (i.e., to check whether a previous verifier obtained a comprehensible result) by instrumenting the produced witness into the verified program. While cooperative methods are effective for reducing the workload of a verification task, employing cooperative methods at tool level would require to exchange analysis information between tools. In general, existing verification tools are not well suited for this type of cooperation, which lead us to explore black-box verifier

combinations. In addition, we showed that non-cooperative methods can improve the verification effectiveness without the need to adapt the employed tools.

Combining Algorithms Beyond Software Verification. The idea of combining algorithms to improve performance have been successfully applied in many research areas including SAT solving [51, 54, 56], constraint-satisfaction programs [21, 45, 57] and combinatorial-search problems [41]. Employed approaches traditionally focused on portfolio-based approaches [21, 51, 54], but recent techniques started to integrate algorithm selectors for either selecting single algorithms [45, 56] or portfolios of algorithms [44, 57]. For example, earlier works in SAT solving [51, 54] focused on parallel-portfolio solvers, while later works such as SATzilla [56] further improves the solving process by selecting a task-dependent solver. However, existing techniques often employ hybrid strategies between portfolios and algorithm selection to achieve state-of-the-art performance. Therefore, Kashgarani and Kothoff [38] have recently shown that parallel portfolios are generally bottlenecked by the available resources and that a pure algorithm selector that selects a single algorithm performs better. While we observed that portfolios of software verifiers are also restricted by available resources (i.e., the performance generally stops to improve after a certain portfolio size), we found that all evaluated combination types yield a similar performance gain when configured correctly.

7 Conclusion

This paper describes a method to construct combinations of verification tools in a systematic and modular way. The method does not require any changes to the verification tools that are used to construct the combinations. Our experimental evaluation shows that all three considered combinations (sequential portfolio, parallel portfolio, and algorithm selection) can lead to performance improvements. The improvements can be significant although the construction does not require significant development effort, because we use CoVeriTeam for the combination and execution of verification tools. We hope that our contribution makes it easy for practitioners to get access to the best performance out of the latest research and development efforts in software verification.

Declarations

Data Availability Statement. A reproduction package including all our results is available at Zenodo [16]. Additionally, the result tables are also available on a supplementary web page for convenient browsing.[7]

Funding Statement. This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 418257054 (Coop).

Acknowledgement. We thank Tobias Kleinert for implementing the parallel portfolio combination in CoVeriTeam.

[7] https://www.sosy-lab.org/research/coveriteam-combinations

References

1. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A framework for abstraction- and interpolation-based software verification. In: Proc. CAV, pp. 672–678. LNCS 7358, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_48
2. Ball, T., Rajamani, S.K.: The Slam project: Debugging system software via static analysis. In: Proc. POPL. pp. 1–3. ACM (2002). https://doi.org/10.1145/503272.503274
3. Beckert, B., Hähnle, R.: Reasoning and verification: State of the art and current trends. IEEE Intelligent Systems **29**(1), 20–29 (2014). https://doi.org/10.1109/MIS.2014.3
4. Beyer, D.: Second competition on software verification (Summary of SV-COMP 2013). In: Proc. TACAS. pp. 594–609. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_43
5. Beyer, D.: Results of the 10th Intl. Competition on Software Verification (SV-COMP 2021). Zenodo (2021). https://doi.org/10.5281/zenodo.4458215
6. Beyer, D.: Software verification: 10th comparative evaluation (SV-COMP 2021). In: Proc. TACAS (2). pp. 401–422. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_24, preprint available.
7. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. LNCS 13244, Springer (2022)
8. Beyer, D., Dangl, M.: Strategy selection for software verification based on boolean features: A simple but effective approach. In: Proc. ISoLA. pp. 144–159. LNCS 11245, Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_11
9. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
10. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
11. Beyer, D., Jakobs, M.C.: Fred: Conditional model checking via reducers and folders. In: Proc. SEFM. pp. 113–132. LNCS 12310, Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_7
12. Beyer, D., Jakobs, M.C., Lemberger, T.: Difference verification with conditions. In: Proc. SEFM. pp. 133–154. LNCS 12310, Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_8
13. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
14. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Combining verifiers in conditional model checking via reducers. In: Proc. SE. pp. 151–152. LNI P-292, GI (2019). https://doi.org/10.18420/se2019-46
15. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. Springer (2022)
16. Beyer, D., Kanav, S., Richter, C.: Reproduction Package for Article 'Construction of Verifier Combinations Based on Off-the-Shelf Verifiers'. Zenodo (2022). https://doi.org/10.5281/zenodo.5812021
17. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16

18. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
19. Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
20. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
21. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Twenty-First International Joint Conference on Artificial Intelligence (2009)
22. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
23. Chalupa, M., Jašek, T., Novák, J., Řechtáčková, A., Šoková, V., Strejček, J.: Symbiotic 8: Beyond symbolic execution (competition contribution). In: Proc. TACAS (2). pp. 453–457. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_31
24. Chong, N., Cook, B., Eidelman, J., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Code-level model checking in the software development workflow at Amazon Web Services. Softw. Pract. Exp. **51**(4), 772–797 (2021). https://doi.org/10.1002/spe.2949
25. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
26. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS. pp. 423–425. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_34
27. Darke, P., Agrawal, S., Venkatesh, R.: VeriAbs: A tool for scalable verification by abstraction (competition contribution). In: Proc. TACAS (2). pp. 458–462. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_32
28. Demyanova, Y., Pani, T., Veith, H., Zuleger, F.: Empirical software metrics for benchmarking of verification tools. Formal Methods in System Design **50**(2-3), 289–316 (2017). https://doi.org/10.1007/s10703-016-0264-5
29. Demyanova, Y., Veith, H., Zuleger, F.: On the concept of variable roles and its use in software analysis. In: Proc. FMCAD. pp. 226–230. IEEE (2013). https://doi.org/10.1109/FMCAD.2013.6679414
30. Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schüssele, F.: Ultimate Taipan with symbolic interpretation and fluid abstractions (competition contribution). In: Proc. TACAS (2). pp. 418–422. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_32
31. Filliâtre, J.C., Paskevich, A.: Why3: Where programs meet provers. In: Programming Languages and Systems. pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
32. Gadelha, M.Y.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: Esbmc v6.0: Verifying C programs using k-induction and invariant inference (competition contribution). In: Proc. TACAS (3). pp. 209–213. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_15

33. Heizmann, M., Chen, Y.F., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate Automizer and the search for perfect interpolants (competition contribution). In: Proc. TACAS (2). pp. 447–451. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
34. Hoare, C.A.R.: The verifying compiler: A grand challenge for computing research. J. ACM **50**(1), 63–69 (2003). https://doi.org/10.1145/602382.602403
35. Holík, L., Kotoun, M., Peringer, P., Šoková, V., Trtík, M., Vojnar, T.: Predator shape analysis tool suite. In: Proc. HVC. pp. 202–209. LNCS 10028 (2016). https://doi.org/10.1007/978-3-319-49052-6_13
36. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **275**(7), 51–54 (1997). https://doi.org/10.1126/science.275.5296.51
37. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
38. Kashgarani, H., Kotthoff, L.: Is algorithm selection worth it? comparing selecting single algorithms and parallel execution. In: AAAI Workshop on Meta-Learning and MetaDL Challenge. pp. 58–64. PMLR (2021)
39. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
40. Kotoun, M., Peringer, P., Šoková, V., Vojnar, T.: Optimized PredatorHP and the SV-COMP heap and memory safety benchmark (competition contribution). In: Proc. TACAS. pp. 942–945. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_66
41. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. In: Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach, pp. 149–190. LNCS 10101, Springer (2016). https://doi.org/10.1007/978-3-319-50137-6_7
42. Kröning, D., Tautschnig, M.: Cbmc: C bounded model checker (competition contribution). In: Proc. TACAS. pp. 389–391. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
43. Lauko, H., Štill, V., Ročkai, P., Barnat, J.: Extending Divine with symbolic verification using SMT (competition contribution). In: Proc. TACAS (3). LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_14
44. Lindauer, M., Hoos, H., Hutter, F.: From sequential algorithm selection to parallel portfolio selection. In: International Conference on Learning and Intelligent Optimization. pp. 1–16. Springer (2015). https://doi.org/10.1007/978-3-319-19084-6_1
45. Minton, S.: Automatically configuring constraint satisfaction programs: A case study. Constraints **1**(1-2), 7–43 (1996). https://doi.org/10.1007/BF00143877
46. Peringer, P., Šoková, V., Vojnar, T.: PredatorHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: Proc. TACAS (2). pp. 408–412. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_30
47. Rice, J.R.: The algorithm selection problem. Advances in Computers **15**, 65–118 (1976). https://doi.org/10.1016/S0065-2458(08)60520-3
48. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x

49. Richter, C., Wehrheim, H.: PeSCo: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS (3). pp. 229–233. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19
50. Richter, C., Wehrheim, H.: Attend and represent: a novel view on algorithm selection for software verification. In: Proc. ASE. pp. 1016–1028 (2020). https://doi.org/10.1145/3324884.3416633
51. Roussel, O.: Description of ppfolio (2011). Proc. SAT Challenge p. 46 (2012)
52. Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: Thread-modular abstract interpretation using side-effecting constraints (competition contribution). In: Proc. TACAS (2). pp. 438–442. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_28
53. Wendler, P.: CPAchecker with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In: Proc. TACAS. pp. 613–615. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_45
54. Wotzlaw, A., van der Grinten, A., Speckenmeyer, E., Porschen, S.: pfoliouzk: Solver description. Proceedings of SAT Challenge p. 45 (2012)
55. Xu, L., Hoos, H.H., Leyton-Brown, K.: Hierarchical hardness models for SAT. In: International Conference on Principles and Practice of Constraint Programming. pp. 696–711. Springer (2007). https://doi.org/10.1007/978-3-540-74970-7_49
56. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. JAIR **32**, 565–606 (2008). https://doi.org/10.1613/jair.2490
57. Yun, X., Epstein, S.L.: Learning algorithm portfolios for parallel execution. In: International Conference on Learning and Intelligent Optimization. pp. 323–338. Springer (2012). https://doi.org/10.1007/978-3-642-34413-8_23

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.