Formal Methods in System Design (2025) 66:99–130
https://doi.org/10.1007/s10703-024-00449-y

Construction of verifier combinations from off-the-shelf components

Dirk Beyer[1] · **Sudeep Kanav**[1] · **Tobias Kleinert**[1] · **Cedric Richter**[2]

Received: 19 November 2022 / Accepted: 7 June 2023 / Published online: 11 March 2025

Abstract
Software verifiers have different strengths and weaknesses, depending on the characteristics of the verification task. It is well-known that combinations of verifiers via portfolio- and selection-based approaches can help to combine their strengths. In this paper, we investigate (a) how to easily compose such combinations from *existing*, 'off-the-shelf' verifiers without changing them and (b) how much performance improvement each combination can yield, regarding the effectiveness (number of solved verification tasks) and efficiency (consumed resources). First, we contribute a method to systematically and conveniently construct verifier combinations from existing tools using CoVeriTeam. We consider sequential portfolios, parallel portfolios, and algorithm selections. Second, we perform a large experiment to show that combinations can improve the verification results *without* additional computational resources. Our benchmark set is the category *ReachSafety* as used in the 11th Competition on Software Verification (SV-COMP 2022). This category contains 5 400 verification tasks, with diverse characteristics. The key novelty of this work in comparison to the conference version of the article is to introduce a validation step into the verifier combinations. By validating the output of the verifier, we can mitigate the adverse effect of unsound tools on the performance of portfolios, especially parallel portfolios, as observed in our previous experiments. We confirm that combinations employing a validation process are significantly more robust against the inclusion of unsound verifiers. Finally, all combinations are constructed from off-the-shelf verifiers, that is, we use the verification tools as published. The results of our work suggest that users of combinations of verification tools can achieve a significant improvement at a negligible cost, and more robustness by using combinations with validators.

Keywords Software verification · Program analysis · Cooperative verification · Tool combinations · Portfolio · Algorithm selection · CoVeriTeam · BenchExec

✉ Dirk Beyer
dirk.beyer@sosy-lab.org
https://www.sosy-lab.org/people/beyer/

Sudeep Kanav
sudeep.kanav@gmail.com

[1] LMU Munich, Munich, Germany

[2] Carl von Ossietzky University, Oldenburg, Germany

1 Introduction

Automatic software verification has been an active area of research since two decades [1], and various tools and techniques have been developed to solve the problem of verifying software [2–8]. The research has also been adopted in practice [9–12]. Each tool and technique has its own strengths in specific areas. In fact, an analysis of the results of category *ReachSafety* of SV-COMP 2022 [5] shows that even though high-performing tools such as CPACHECKER [13, 14] and ESBMC [15] share tasks that both tools can solve, there is a significant number of tasks solved uniquely by one of the two tools (see Fig. 1). 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 verification tasks (e.g., up to 692 tasks more for a combination of CPACHECKER and ESBMC). Most current combination approaches are hard-coded, that is, the choice of the tools to combine is fixed and the glue-code required 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. We considered the following three types of combinations: sequential and parallel portfolio [16], and algorithm selection [17]. In a sequential portfolio, the components are executed in sequence (one after another) until one of them succeeds (split time; full cores and memory; split risk). In a parallel portfolio, the components are executed in parallel until one of them solves the task (full time; split cores and memory; split risk). In algorithm selection, first, a selector selects a component that is most likely to solve the given task, and then only the chosen component is executed (full time; full cores and memory; full risk).

We use COVERITEAM [18–20] to construct and execute the combinations. 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.

This paper is an extended version of an article presented at FASE 2022 [21], with an attempt to mitigate its limitations. One of the limitations was that parallel portfolios are biased toward faster tools and would produce incorrect results if there is a fast but unsound tool included in the portfolio. We had mentioned two remedies for this issue: (i) add a validation step after the verification and (ii) carefully select the verifiers to include in a portfolio.

In this work, we apply the first remedy: we add a validation step to validate the results produced by a verifier. The verifiers that we use in our experiments also produce witnesses,

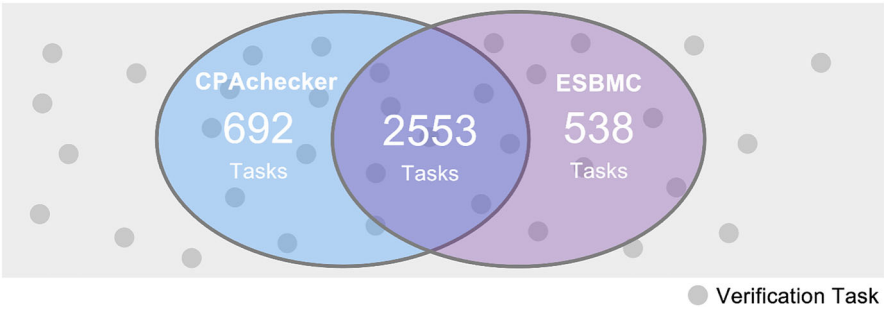

Fig. 1 Overlap of tasks solved by CPACHECKER and ESBMC in SV-COMP 2022

in addition to the verdict, if the verification succeeds. Producing witnesses is a requirement to participate in SV-COMP, as the competition grants points only to those verifiers whose results can be validated. We use a subset of validators used in SV-COMP. For each VERIFIER we construct a VERIFIER$^{\text{Val}}$: a sequence of a verifier and a validator. We then put this combination in sequential portfolios, parallel portfolios, and algorithm selections.

Our experiments are based on tools and benchmark verification tasks from the 11th Competition on Software Verification (SV-COMP 2022) [5].

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 combination approaches based on sequential portfolios, parallel portfolios, and algorithm selections.
3. We provide a reproduction package containing tools and experiment data [22].

2 Overview of Combination Types for Off-the-Shelf Verifiers

In this study, we explore different types of combining verifiers to improve the overall verification effectiveness. We focus on the most common types of combinations that do not require any changes to the existing tools (off-the-shelf) or communication between the tools, which are: *sequential portfolio* [13, 23, 24], *parallel portfolio* [16, 25, 26], and *algorithm selection* [17, 27–30]. We now briefly describe these combination types and give an illustration in Fig. 2.

Sequential Portfolio. Portfolios combine several verifiers by executing them either sequentially or in parallel. A sequential portfolio (Fig. 2a) executes a set of verifiers in sequence, running them one 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 can solve the given verification task within the allocated time, the portfolio is stopped and the solution is emitted. Otherwise, if the current verifier runs into a timeout without solving the given task, it is terminated and the next one is started. CPA-Seq [13, 23] and Ultimate Automizer [24] are examples of a sequential portfolio.

Parallel Portfolio. In contrast to a sequential portfolio, a parallel portfolio (Fig. 2b) executes all verifiers in parallel, sharing all system resources like CPU cores and memory. As soon as one algorithm solves the given verification task, the portfolio is stopped and the solution is emitted. Since the physical computing resources are shared in a parallel portfolio, a tool may use up all its memory quota sooner than when running alone, and be terminated. PredatorHP [25, 26] is an example of a parallel portfolio.

Algorithm Selection. To reduce spending resources on unsuccessful verifiers, algorithm selection (Fig. 2c) is designed to select the verifier that is likely well suited to solve the given verification task. More precisely, algorithm selection first analyzes the given verification task for common characteristics, e.g., program features like the existence of a loop or an array. It then selects a verifier that is most likely to solve verification tasks with those characteristics. Then the selected verifier is executed. Algorithm selection was recently explored for selecting from a set of verification algorithms, e.g., in PeSCo [27, 28], or from a set of sequential portfolios of verification algorithms, e.g., in CPAchecker [29].

The above combination types have their own advantages and limitations when applied in real-world scenarios. While algorithm selection gives the full resources to one verifier, and thus, increases the chances that the verifier succeeds, it also takes the full risk of selecting

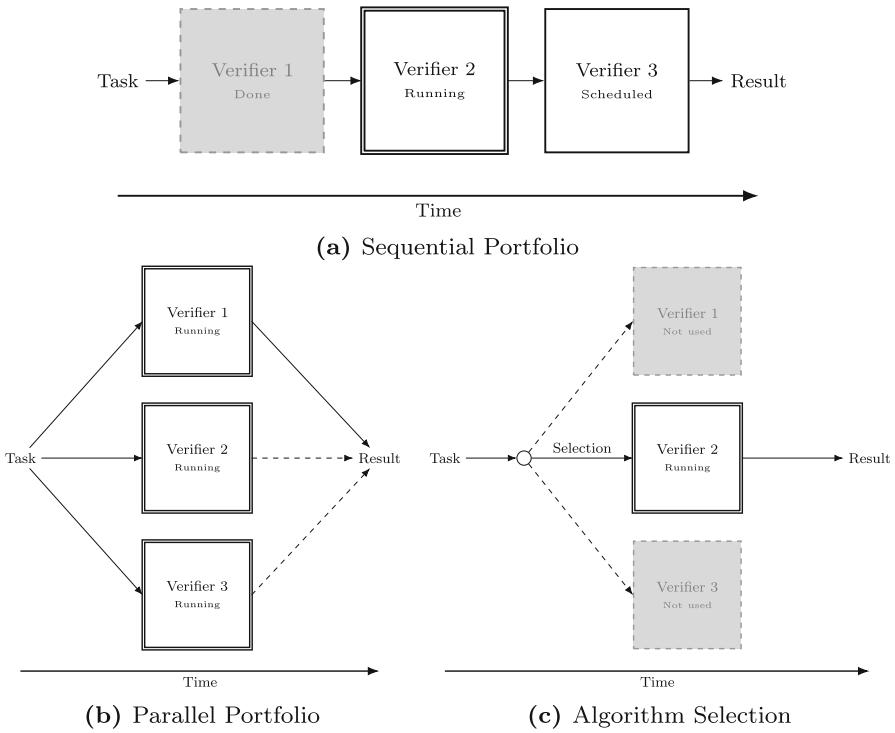

Fig. 2 (**a**) A sequential portfolio runs each verifier for a certain maximal amount of time in a sequence. If a verifier stops with a result, the portfolio finishes. The available CPU time is split among the verifiers. (**b**) A parallel portfolio runs all verifiers simultaneously. If a verifier stops with a result, the portfolio finishes. The available CPU cores and memory are split among the verifiers. (**c**) An algorithm selection first selects a verifier and then executes it. The result produced by this verifier is taken. The selected verifier gets all available resources, and also the risk that the verifier does not deliver a result is not split

a sub-optimal verifier. If the selection algorithm is not powerful enough or the selection task is too difficult (i.e., the selection cannot be decided based on high-level features), the selector might fail to identify a verifier that is appropriate for the given task. Although portfolios omit this problem by assigning the verification task to several verifiers, each verifier gets fewer resources.

Validation of the Verifier Results. Verifiers can have bugs, hence, it is desirable to validate the result of the verification. One of the proposed options in the literature is that the verifier produces a justification of its verdict in the form of a verification witness in an exchangeable format [31–34]. A user can either inspect the witness manually [35] or use a tool to validate the result produced by the verifier.

Verification followed by result validation can also be achieved by a combination of a verifier and a validator. First, a verifier is executed to solve a verification task, then a validator is executed to validate the result of the verifier. We refer to such a combination as *validating verifier* (or VERIFIER$^{\text{Val}}$). A validating verifier has the advantage that each emitted verification result is validated by an external validator.

Therefore, we can avoid emitting wrong results, which in turn can positively impact the effectiveness of verifier combinations.

Validators have been used in the competition on software verification since 2015 [36]. Intuitively, one can expect that the validation of a bug consumes much less resources compared to finding the bug, whereas the proof validation is still resource intensive because the full state-space must be considered. This intuition is supported by the resource-consumption data of the validators in the software-verification competitions. Moreover, in the competition, alarm validation is allocated 10 % (= 1.5 min) of the CPU time of a verification run (15 min), whereas proof validation gets the same CPU time as verification. Both the alarm and proof validation are allocated about half the memory as verification (7 GB).

3 Constructing Combinations with CoVeriTeam

CoVeriTeam [18] is a tool for creating and executing tool combinations. It consists of a language to describe combinations of tools and an execution engine for their execution. Tools, e.g., verifiers, validators, testers, transformers, and their combinations, are called *verification actors* in CoVeriTeam. The inputs consumed and outputs produced by the verification actors, e.g., programs, specifications, witnesses, and results, are called *verification artifacts*. Verification artifacts are seen as basic objects, verification actors as basic operations, and tool combinations as compositions of these operations.

Verification actors in CoVeriTeam are of two kinds: *atomic* and *composite*. Atomic actors are based on off-the-shelf tool archives. CoVeriTeam uses features provided by BenchExec [37] to configure the command line, execute the tool in isolation, enforce resource limits, and process the output produced by the tool. Atomic actors are constructed using the information provided in a YAML configuration file, which specifies the BenchExectool-info module, parameters to pass to the tool, resource limits, and the location to download the tool archive from. Many publicly available tools for automatic verification are supported by CoVeriTeam, and their YAML configuration files are available in the CoVeriTeam repository [19].

Composite actors are created by combining CoVeriTeam actors using the following composition operators: SEQUENCE, PARALLEL, REPEAT, ITE, and PARALLEL-PORTFOLIO. SEQUENCE executes the composed actors sequentially, PARALLEL in parallel, REPEAT repeatedly until the termination condition is satisfied, ITE (*if-then-else*) executes one actor if the provided condition is true, otherwise, the other actor, PARALLEL-PORTFOLIO executes the actors in parallel until one of them finishes with a result that satisfies the success condition, and then terminates all remaining actors [18, 21, 38]. The work in this paper uses SEQUENCE, PARALLEL, ITE, and PARALLEL-PORTFOLIO.

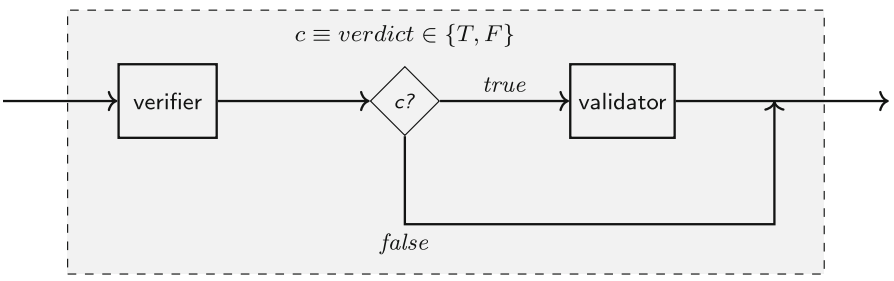

Fig. 3 Construction of a VERIFIER$^{\text{Val}}$ using CoVeriTeam

3.1 Validating Verifier

Figure 3 shows the construction of a combination VERIFIER[Val] for a verifier verifier and a validator validator. This construction uses the CoVeriTeam composition operators SEQUENCE and ITE. The combination first executes the verifier. Then, it checks whether the verifier solved the verification task (i.e., finishes with a verdict *true* or *false*) or not. A verifier might not always succeed; it can finish the execution with an error or be terminated when it runs out of resources. If the verifier solved the verification task, then the combination executes the validator on the result of the verifier, otherwise, the verdict *error* or *unknown* is forwarded. This construction can be generalized to use combinations of validators as validator. For our experiments (see Sect. 4.3), we created validating verifiers using a portfolio of validators instead of one validator.

3.2 Verifier Based on Sequential Portfolio

Figure 4a shows the construction of a sequential portfolio for two verifiers verifier1 and verifier2. This construction is similar to VERIFIER[Val], with the difference that the second actor in this combination is also a verifier instead of a validator. This construction uses the CoVeriTeam composition operators SEQUENCE and ITE. The combination first executes verifier1. If the execution of verifier1 successfully solves the task, then the combination finishes with the result of verifier1, otherwise, verifier2 is executed and the combination finishes with the result of verifier2.

This construction can be generalized to create sequential portfolios of arbitrary sizes. For our experiments, we created sequential portfolios of 2, 3, 4, and 8 verifiers.

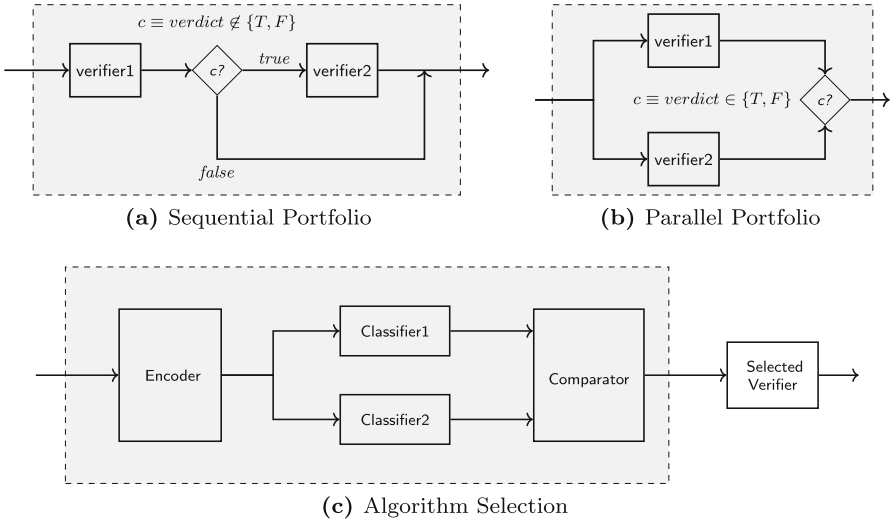

Fig. 4 Construction of verifiers based on sequential portfolio, parallel portfolio, and algorithm selection using CoVeriTeam(the incoming arrow from the left always represents the verification task; the outgoing arrow on the right always represents the verification result)

3.3 Verifier Based on Parallel Portfolio

Figure 4b shows the construction of a parallel portfolio for two verifiers verifier1 and verifier2. This construction uses the CoVeriTeam composition operator PARALLEL-PORTFOLIO, which combines a set of actors of the same type (verifiers, testers, etc.) based on a success condition. The success condition is defined over the artifacts produced by these actors, and is evaluated whenever an actor finishes its execution. In this combination, both the verifiers are executed concurrently. When one verifier finishes, its verdict is checked for the success condition (i.e., $verdict \in \{T, F\}$). If the success condition holds, then the combination finishes (terminating all remaining executing verifiers) and returns the verdict, otherwise, the verdict is discarded and the combination waits for the second verifier to finish. If none of the verifiers produces a result that satisfies the success condition, then the combination returns the result of the last verifier. This construction can be generalized to create parallel portfolios of arbitrary sizes. For our experiments, we created parallel portfolios of 2, 3, 4, and 8 verifiers.

3.4 Verifier Based on Algorithm Selection

Figure 4c shows the construction of an algorithm selection for two verifiers verifier1 and verifier2. This construction uses the CoVeriTeam composition operators SEQUENCE and PARALLEL, and some CoVeriTeam actors for feature encoding, classifiers, and comparator. The combination consists of two parts, the selector to determine an appropriate verifier based on the given verification task and the execution of the selected verifier. In more detail, the combination first executes the feature encoder on the verification task, in which a set of predefined features is extracted and encoded from a given verification task (i.e., certain characteristics that are believed to indicate difficulty for a verifier). The output is passed on to a set of classifiers (classifier1 and classifier2), one for each verifier that is considered for selection. Each classifier predicts the hardness (or difficulty) of the given verification task for the corresponding verifier. The comparator then compares the hardness scores and determines the verifier with the least value, which is predicted to be the most appropriate verifier for the given verification task based on the extracted features, i.e., the verifier that is most likely to solve the task. The last step is to execute only the verifier that was selected (for example, execute verifier1, do not execute verifier2). This construction can be generalized to create algorithm selections of arbitrary sizes. For our experiments, we created algorithm selections of 2, 3, 4, and 8 verifiers.

Feature Encoder. The first component of our construction is the feature encoder. The goal of the feature encoder is to encode the verification task into a meaningful feature-vector (FV) representation that can later be used 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 [30] or variable types [39, 40].

In this study, we encode verification tasks via a learning-based feature encoder by employing a pre-trained *CSTTransformer* [28]. The CSTTransformer first parses a given program P into a simplified abstract syntax tree (AST) representation. Afterward, a graph-based neural network processes the AST structure and produces a vector representation. The encoding step is learned by pre-training the neural network on selecting various verification tools. While this approach was originally developed to learn a vector representation optimized for a specific verifier combination, the authors have shown 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. One verification tool might be able to solve a given verification task quickly, whereas another tool might fail to solve it even using all given resources. 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], we learn to predict the difficulty of a task with *hardness models* [41]. A hardness model learns to predict the difficulty, also called hardness, of a given task for a specific tool based on the previously computed vector representation of the task. In our case, we define the hardness of a task for a given tool similar to the PAR10 score [42]: It is either the consumed CPU time if the task is solved correctly or ten times the maximal runtime. A low hardness score means that a verifier solves a task correctly in a short amount of time.

Since our hardness score is based on the CPU time consumed by the verifier, the problem of training our hardness model reduces to a regression problem. We address this problem with regression by classification [43] by training multinomial logistic-regression classifiers.

Given a set of hardness models—each assessing 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.

3.5 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 requires only an actor definition for that verifier in CoVeriTeam. Afterwards, this verifier can be easily added to a sequential or parallel portfolio.

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, the task of configuring algorithm selection has been simplified by using hardness models together with a common feature representation.

One can modify the set of verifiers to select from by simply adding or removing individual hardness models. While previous approaches to verifier selection often require training the complete selector from scratch, our combination can be extended by training a single hardness model. A single hardness model can be trained within a few minutes on a modern CPU. The accompanying artifact contains all the training scripts that we used for training our hardness models, a pre-computed dataset of vector representations for SV-COMP 2022, and instructions to train a new model. It is also possible to train and employ custom hardness models based on a custom vector representation. In this case, one needs to replace the feature encoder, which can easily be done as it is a CoVeriTeam actor in our construction.

Finally, to integrate a new tool in our algorithm selector, one is only required to run the respective verifier once on (a subset of) the benchmark set. The results then act as training examples.

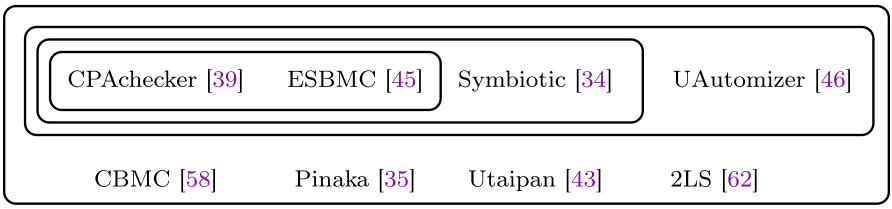

Fig. 5 Subsets of 2, 3, 4, and 8 verification tools as used in our combinations

4 Experiment Setup

Our goal is to investigate if combinations can yield better results than standalone tools. To achieve this goal, we have chosen the following measures for comparison: number of solved verification tasks, normalized score[1], and resource consumption. For each combination, we compared the best-performing standalone verifier against the combination using these measures. We derived the following three research questions from our research goal:

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,
(b) normalized score, and
(c) 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,
(b) normalized score, and
(c) 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,
(b) normalized score, and
(c) resource consumption?

To address the above research questions, we performed an extensive experimental evaluation. This section explains the setup of our experiment.

4.1 Selection of Existing Verifiers

We considered the results of the Competition on Software Verification 2022 [5] for selecting the verification tools for our combinations. We chose the 8 best tools from the ReachSafety category, and sorted them according to their scores in SV-COMP 2022. Then we took the top n tools for a combination of size n. Figure 5 illustrates the sets of verifiers that we used in different types of combinations.

[1] The benchmark set is partitioned into categories of different sizes. The number of solved verification tasks is biased towards performance in large benchmark sets. Using a normalized score mitigates this bias.

Exclusions. We excluded the following verification tools from consideration: VERIABS [49], because its license does not allow us to use it for scientific evaluation, PESCO [50], because it would not contribute to the diversity of technologies in the combinations as it is based on CPACHECKER configurations, GRAVES- CPA [51], for the same reason as PESCO, and COVERITEAM- VERIFIER- PARALLELPORTFOLIO and COVERITEAM- VERIFIER- ALGOSELECTION, because they are themselves combinations of verifiers similar to the ones we evaluate in this paper.

4.2 Selection of Existing Validators

We chose the validators also based on the results of the competition on software verification 2022 [5]. We took the validators that were most effective in validating witnesses in the competition as reported in a case study [52]. We took four validators for violation witnesses (*alarm validation*): CPACHECKER-based violation-witness validator [32], FSHELL-WITNESS2TEST [34], SYMBIOTIC- WITCH [53], and NITWIT [54]; and two validators for correctness witnesses (*proof validation*): CPACHECKER- and UAUTOMIZER-based correctness-witness validators [33]. We have excluded METAVAL [55], because it was not adopted to a new rule of SV-COMP 2022 [52].

4.3 Construction of VERIFIER$^{\text{Val}}$ Combinations

Figure 6 shows the construction that validates the results of a verifier. We have used portfolios of validators of different sizes: 2 for validating proofs, and 4 for validating alarms.

The combination first executes the verifier. Then, if the produced *verdict* is *true*, it executes the portfolio of *proof validators*. If the produced *verdict* is *false*, it executes the portfolio of *alarm validators*. If the produced verdict is neither *true* nor *false*, i.e., the verifier did not succeed in its verification effort, then the verdict *error* or *unknown* is simply forwarded. The proof and alarm validators are combined in parallel portfolios, respectively. (The figure shows a simplified presentation of the combination, using a verdict check with

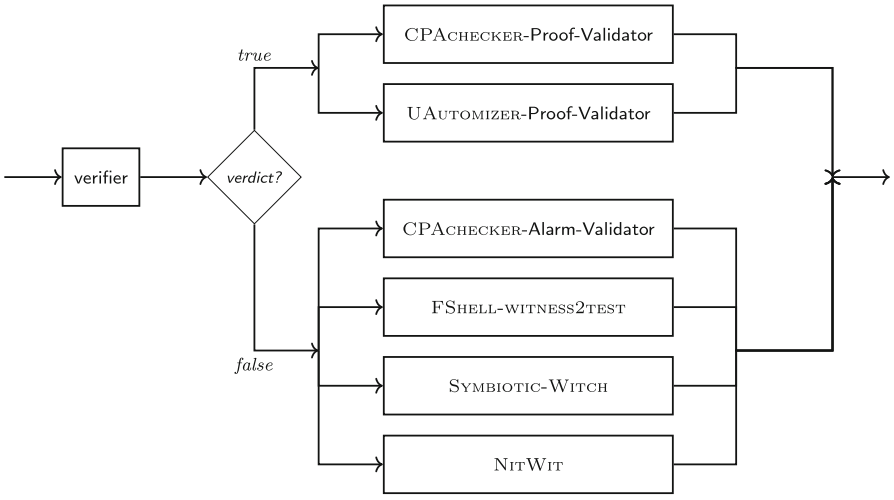

Fig. 6 Construction of our VERIFIER$^{\text{Val}}$ combination

three outcomes instead of two ite operators, and omitting the check of success conditions in parallel portfolios.)

We instantiated the VERIFIER$^{\text{Val}}$ combination for all eight selected verifiers and refer to them as *single verifiers* because they are combined only with validators. We executed the resulting combinations on the chosen benchmark set. The obtained results were also used to train the hardness models used by the algorithm selector.

4.4 Training of the Algorithm Selector

Our selection mechanism is based on the use of hardness models. We trained several hardness models that predict the difficulty of a given verification task for a specific verifier. The algorithm selector then selects the verifier that is most likely to solve a given task, i.e., for which the task is the easiest. In the following, we describe the process for training the individual hardness models used in our evaluation in detail.

Construction of Training Datasets. Hardness models learn from prior observations of the verifier's performance to predict the difficulty of future tasks. Therefore, we trained the hardness models on a dataset of verification tasks labeled with the results of the individual verifiers.

We employed a random *subset* of the benchmark set used in SV-COMP 2022 [5] as the training dataset of verification tasks. Therefore, the training dataset partly overlapped with the benchmark set (up to 90 %) used in our evaluation. We maintain a fair comparison between algorithm selectors by training them on the same train/test split. To obtain the results of the individual verifiers on the verification tasks, we executed all the single verifiers on the benchmark set. We recorded whether the verifier solves the task *correctly* and the *execution time* in CPU seconds.

Finally, since our hardness models operate on feature-vector representations, we employ our feature encoder to map each verification task to a feature vector. As a result, we obtain n datasets (n is the number of verifiers) where each entry maps a feature-vector representation of a verification task to the correctness and execution time of a verifier on that task.

Training Hardness Models. The hardness models are trained to predict the hardness of a task for a given verifier. Similar to the PAR10 score, we define the hardness score (h-score) of a given task for a specific verifier as the CPU time if the task can be solved within a certain time limit, or ten times the time limit if the task cannot be solved. For the prediction, we split the range of our h-scores into four intervals[2]: [0, 10), [10, 100), [100, 900), [900, 9000]. If the verifier solves the task correctly, the hardness model predicts whether solving the task was easy ([0, 10)), intermediate ([10, 100)) or difficult ([100, 900)). In the case that the verifier fails, the hardness model should predict that the task was too hard ([900, 9000]).

Motivated by the idea of regression by classification [43], we address this problem by training a multinomial-logistic-regression classifier. Then, for each interval, the classifier predicts the probability that it contains the h-score of the verifier for the given task. Finally, to obtain a predicted hardness score which we can use to select a verifier, we make the following observation for the hardness score:

$$\text{h-score}(x) \leq \sum_{i=0}^{k} p\big(\text{h-score}(x) \in [l_i, u_i)\big) * u_i,$$

where x is the given verification task, k is the number of disjoint intervals $[l_i, u_i)$. In other words, if we can correctly estimate the probability p of a hardness score to be included in an

[2] The verification timeout is typically set to 900 s and we found splitting intervals logarithmically works best.

interval $[l_i, u_i)$, we can compute an upper bound to the hardness score. We train the logistic-regression classifier to estimate the probability and use the upper bound as the predicted hardness score to select a verifier, i.e., we select the verifier whose hardness score is likely bounded by the smallest constant. We compared our approach with alternative approaches that predict the likelihood of solving an instance [28], solvability and runtime independently [30], or the hardness score via linear-regression models. However, in our experiments, we found that predicting hardness-score intervals leads to the best algorithm-selection performance.

Selecting a Verifier. After training, the hardness models predict the difficulty of a given task for a specific verifier. Given a new verification task, the feature encoder is executed followed by the classifiers (hardness models). As a result, we obtain a hardness score for each verifier in our combination. Then, the verifier obtaining the smallest hardness score is selected and executed.

Since the predictions are made independently, our algorithm-selection framework is modular. In other words, we can simply extend or shrink the size of the combination by adding or removing verifiers and their respective hardness models.

4.5 Construction of Portfolio and Selection Combinations

We evaluated twelve verifier combinations. For each sequential portfolio, parallel portfolio, and algorithm selection, we constructed a combination of 2, 3, 4, and 8 verifiers. This gave us four combinations for each combination type. These variants of combinations with different numbers of verifiers allowed us to quantify the influence of the number of verifiers on the performance.

4.6 Resource Allocation

Resource Allocation for Actors inside VERIFIER$^{\text{VAL}}$. Fig. 7 shows the resource limits for the actors inside a VERIFIER$^{\text{Val}}$ composition. Given the resource limits T (time) and M

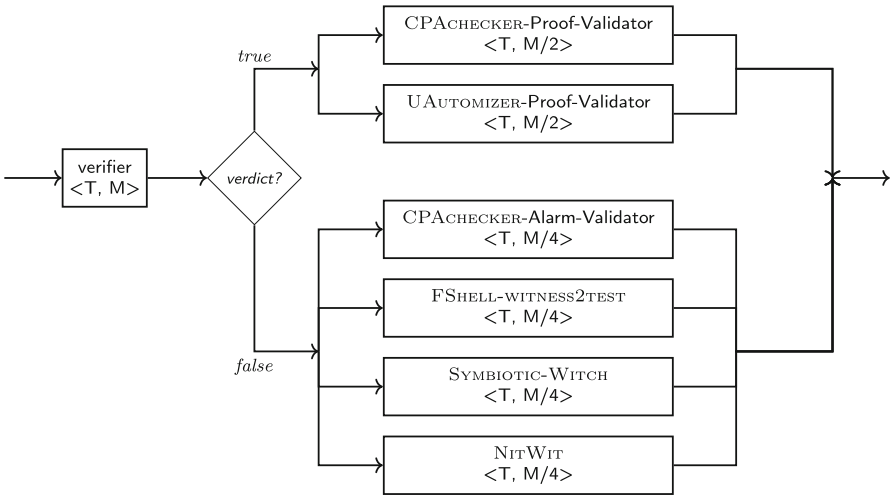

Fig. 7 Resource allocation for the VERIFIER$^{\text{Val}}$ combination

(memory) for the complete VERIFIER^{Val}, we define the resource limits for the actors in the VERIFIER^{Val} composition as follows:

- Verifier <T, M>: We give the same resource limits to the verifier. It is the first actor to be executed in the combination. As no other tool executes when the verifier is executing, it is fair to allow the verifier to use as much memory as is available. Moreover, as the validation can only start after the verifier finishes execution and produces a result, we give it all the available time.
- Proof-Validator <T, M/2>: We divide the available memory but pass on the same time limit to the proof-validator. During the validation stage, two proof-validators are running simultaneously. To allow fair distribution of memory we divide the memory limit equally among the two proof-validators. We do not divide the CPU time because this combination is a parallel portfolio of validators and we do not limit CPU time in parallel portfolios.
- Alarm-Validator <T, M/4>: Analogous to the proof-validators, we divide the available memory but pass on the same time limit to the alarm-validators.

Resource Allocation for the Combinations of VERIFIER^{Val}. The resource limits of the combinations of single verifiers (of the form VERIFIER^{Val}) to a sequential portfolio, parallel portfolio, or algorithm selection are as follows:

- Sequential portfolio: We divide the available CPU time equally among all actors in a sequential portfolio, but allow them to use all the available memory.[3]
- Parallel portfolio: We divide the available memory equally among all actors in a parallel portfolio, but allow them to use all the available CPU time.
- Algorithm selection: We pass on the available resource limits to the feature encoder and the verifier. For classifiers, we divide the memory limit equally and give them a constant time limit of 20 s. We enforce resource limits on the classifiers because classifiers in our experiments take just a couple of seconds to execute. If one of them behaves unexpectedly and consumes too much time, we proceed without waiting for its execution to finish. We select the verifier to execute based on the results of the remaining classifiers.

4.7 Benchmark Selection

We evaluated the tool combinations on a benchmark set from the open-source collection of verification tasks [56]. The benchmark sets for SV-COMP are also selected from this collection. Our benchmark set consisted of all the verification tasks in the category REACHSAFETY used in SV-COMP 2022. It is the largest category, contains 5 400 verification tasks, and is the most popular one with 21 participants in SV-COMP 2022. Each verification task consists of a program written in C and a specification. The specification is a safety property describing that an error function should never be called. Thus, we had a total of 5 400 verification tasks in our benchmark set. We evaluated our combinations on the version of the benchmark set that was used in SV-COMP 2022 (tag `svcomp22`) [57].

[3] Technical detail: We assign a little bit less memory to the actors of the sequential portfolio because otherwise, if one of them starts consuming more memory than the provided memory limit, then it would make the complete sequential portfolio exceed the memory limit. This would in turn trigger BENCHEXEC to terminate the complete sequential portfolio.

4.8 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, and 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 2022 allowing us to use the competition results of the standalone tools for comparison.

4.9 Scoring Schema

We report three measures of success for each combination. First, we count the number of results of each kind, i.e., either claims of program correctness or alarms of specification violations for the verification tasks.

Second, we report the scores as per the scoring scheme used in the competition SV-COMP [5]. A verifier is rewarded with score points as follows: 2 score points for each correct proof, 1 score point for each correct alarm, -32 score points for each wrong proof, and -16 score points for each wrong alarm. We have used this scoring scheme because it is accepted in the community as a model of quality. The benchmark set is partitioned into several sub-categories, and we calculate the score for each sub-category and apply normalization as in SV-COMP based on the size of the sub-category. The normalization of scores has been used in SV-COMP [58] for many years and has been established as a standard for judging the quality of results by the verification community.

Although the inclusion of a validation step alleviates the issue of using an unsound verifier to a large extent, it does not eliminate it, because validators could also be unsound and could validate incorrect witnesses. Due to this reason, we still need to use negative scores as well.

The scoring scheme used in our previous work [21] employed the same reward scheme but it did not consider normalization. This resulted in scores being biased towards tools that perform well in the larger sub-categories of the benchmarks. In this article, we report the normalized scores addressing the issue of bias towards the results of the large sub-categories of the benchmarks. We have used the same scripts that were used in SV-COMP to calculate these scores.

4.10 Resource Measurement and Benchmark Execution

We used the state-of-the-art benchmarking framework BENCHEXEC [37] for executing our benchmarks. It executes tools in isolation, reports the resource consumption, and enforces the resource limitations. It provides measurements of the consumption of CPU time, wall time, memory, and CPU energy during the execution of a tool.

4.11 Reporting Results: Tables and Plots

We present a table and two plots for each set of experiments.

Tables. We report the normalized score, correctly solved instances of both proofs and alarms, total resource consumption, median resource consumption, and resource consumption per score point for each executed combination.

The tables report the resource consumption only for the correctly solved tasks. It is similar to how the results are presented in the competition reports [5, 59]. Using this approach encourages verifiers to try as hard as possible to solve a verification task without worrying about how it affects the resource consumption. Otherwise, one could, in principle, considerably improve this measure by simply terminating early with the output UNKNOWN.

Score-Based Quantile Plots. For each set of experiments, we also present quantile plots based on the normalized score. Each point (x, y) in these plots represents the score x accumulated for the executions that finished below the CPU time y. The CPU time consumption of only those executions that produce a correct result is considered. The time consumption of executions producing incorrect and inconclusive verdicts are not considered. These plots use a linear scale for the CPU time range between 0 and 1 s, and a logarithmic scale for 1 s to 1 000 s.

Interpretation: The higher the score of a tool, the farther on the right its plot goes. As our scoring scheme penalizes incorrect results, the abscissa of the starting point of each plot line is the total penalty a tool has received. The more unsound a tool is, the farther on the left its plot graph starts. The length of the projection of the plot graph on the horizontal axis loosely corresponds to the total number of correctly solved tasks (because 2 points are awarded for a correct proof, and 1 point for a correct alarm). The height of a plot represents the maximum time required by the corresponding tool to correctly solve a verification task. The area under the graph loosely corresponds to the total time taken by the tool for the executions that resulted in correct results. In essence: the plot graph of a sound, effective, and efficient tool would start at zero on the x-axis, go far towards the right, and remain low. More details about these plots are given in [58].

Parallel-Coordinates Plots. In addition to the tables and quantile plots, we present parallel-coordinates plots for showing the resources consumed per score point. Parallel-coordinates plots are used to display multivariate data points, where each variable gets its own axis and each graph represents one data point. They provide a visual aid to compare many variables and see the relation between them.

Another possibility to show the resource consumption per score point was to use spider charts (also known as radar or web charts). In a spider chart, linear differences in values of a variable scale to a quadratic change in the area, which may give an incorrect impression to a viewer. Therefore, we chose to use parallel-coordinate plots instead of spider charts.

The plots show resource consumption per score point, as well as the number of unsolved tasks per score point. The lower the plot graph remains the better it is.

5 Evaluation Results

5.1 Results for VERIFIER$^{\text{Val}}$ Standalone Compositions

Table 1 shows the summary of results for executing the VERIFIER$^{\text{Val}}$ compositions, that is, combinations of an existing verifier with a validator portfolio as a standalone composition (referred to as *single verifiers*). The scores and ranking are roughly comparable to the results of SV-COMP 2022.[4] Introducing the validation step decreased the scores by about 10 %. Figure 8 shows the score-based quantile plot of the results, and Fig. 9 shows the parallel-coordinates plot for unsolved tasks and resource consumption per score point.

[4] https://sv-comp.sosy-lab.org/2022/results/results-verified.

Table 1 Single verifiers: comparison of results of VERIFIER^{Val} (ordering of columns is based on the score of the underlying verification tools in SV-COMP 2022)

Verifier	CPACHECKER^{Val}	ESBMC^{Val}	SYMBIOTIC^{Val}	UAUTOMIZER^{Val}	CBMC^{Val}	PINAKA^{Val}	UTAIPAN^{Val}	2LS^{Val}
Score, normalized	**5 016**	4 641	4 025	3 513	3 575	3 332	3 506	3 350
Correct results	**3 129**	2 818	2 338	1 997	1 852	1 755	1 792	1 768
Correct proofs	**1 574**	1 453	8 53	1 215	743	845	1 194	1 186
Correct alarms	**1 555**	1 365	1 485	782	1 109	910	598	582
Wrong results	2	**0**	**0**	2	1	**0**	2	**0**
Wrong proofs	**0**	**0**	**0**	2	**0**	**0**	2	**0**
Wrong alarms	2	**0**	**0**	**0**	1	**0**	**0**	**0**
Total resource consumption for correct results								
CPU time (h)	170	67	38	78	39	**26**	63	46
Wall time (h)	110	42	26	35	24	**12**	32	28
Memory (GB)	4 900	3 000	2 200	3 400	**1 600**	2 300	2 100	1 900
CPU Cpuenergy (KJ)	6 200	2 300	1 300	2 400	1 300	**790**	2 000	1 300
Median resource consumption for correct results								
CPU time (s)	150	45	24	86	**22**	31	85	54
Wall time (s)	86	13	13	27	**8.0**	11	25	15
Memory (MB)	1 100	680	**530**	820	550	610	780	710
CPU Cpuenergy (J)	1 500	310	230	640	**170**	250	620	390
Resource consumption of correct results per score point								
CPU time (s/sp)	120	52	34	80	39	**28**	65	50
Wall time (s/sp)	78	33	23	36	24	**13**	33	30
Memory (MB/sp)	980	660	540	960	**450**	680	600	570
Cpuenergy (J/sp)	1 200	500	330	690	370	**240**	580	400

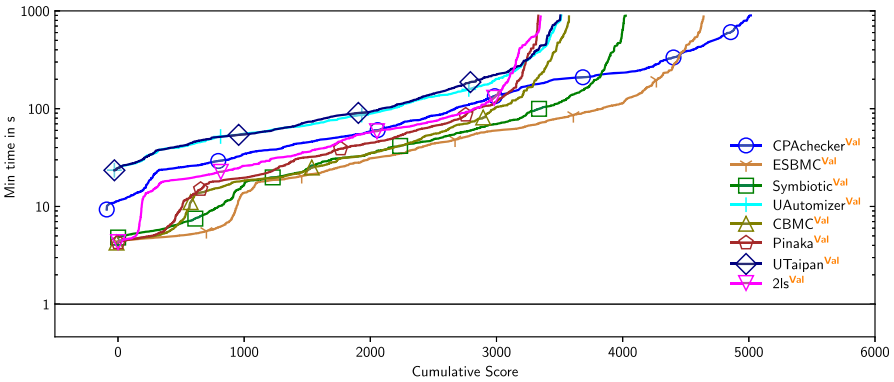

Fig. 8 Single verifiers: Score-based quantile plot for results of VERIFIER[Val] combinations

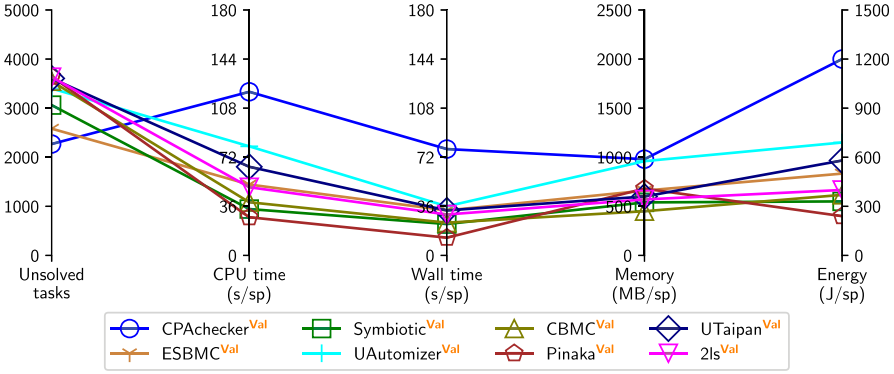

Fig. 9 Single verifiers: Unsolved tasks and resource consumption per score point of VERIFIER[Val] combinations

5.2 RQ 1: Evaluation of Sequential-Portfolio Verifiers

We now present the results of the sequential-portfolio verifiers against the standalone VERIFIER[Val] combination with the highest score: CPACHECKER[Val].

Table 2 shows the summary of results for the sequential portfolio. Three of our sequential portfolios achieve a better score than CPACHECKER[Val]. The portfolio with 8 tools performs worst, which is expected because the amount of time allocated to each verifier decreases as we increase the size of the portfolio. As a result, verifiers cannot solve hard tasks that take long to solve. The table also shows that the portfolios require more resources to solve the tasks. This is a side effect of the sequential portfolio, as all the resources consumed by unsuccessful attempts by the verifiers in a sequence are still counted towards the overall resource consumption.[5]

Figure 10 shows the quantile plot of normalized scores for the best and worst performing sequential portfolios, and CPACHECKER[Val]. All graphs start from the same abscissa because all of them have the same number of incorrect results (a negligible value of 2). The sequential portfolio of 4 tools goes farthest to the right because it has the highest score. Figure 11 shows

[5] This may change with a change in the order of tools in the sequence. One could try to come up with an optimum order by analyzing the results of the standalone CPACHECKER[Val]. But we kept our approach simple and put the tools in the order of the SV-COMP scores.

Table 2 Sequential portfolios: comparison different sizes with CPACHECKER^{Val}

Verifier	CPACHECKER^{Val}	Sequential Portfolio of			
		2	3	4	8
Score, normalized	5 016	5 224	5 230	**5 265**	4 984
Correct results	3 129	3 239	**3 298**	3 292	3 048
Correct proofs	1 574	**1 621**	1 587	1 601	1 385
Correct alarms	1 555	1 618	**1 711**	1 691	1 663
Wrong results	**2**	**2**	**2**	**2**	**2**
Wrong proofs	**0**	**0**	**0**	**0**	**0**
Wrong alarms	**2**	**2**	**2**	**2**	**2**
Total resource consumption for correct results					
CPU time(h)	170	190	190	190	**120**
Wall time (h)	110	130	130	130	**88**
Memory (GB)	4 900	5 400	5 600	5 300	**4 400**
CPU Energy (KJ)	6 200	7 000	7 200	7 300	**4 900**
Median resource consumption for correct results					
CPU time(s)	150	160	170	160	**120**
Wall time (s)	86	96	110	110	**77**
Memory (MB)	1 100	1 100	1 100	1 000	**950**
CPU Energy (J)	1 500	1 600	1 800	1 700	**1 300**
Resource consumption of correct results per score point					
CPU time (s/sp)	120	130	130	130	**90**
Wall time (s/sp)	78	88	93	92	**64**
Memory (MB/sp)	980	1 000	1 100	1 000	**890**
CPU Energy (J/sp)	1 200	1 300	1 400	1 400	**980**

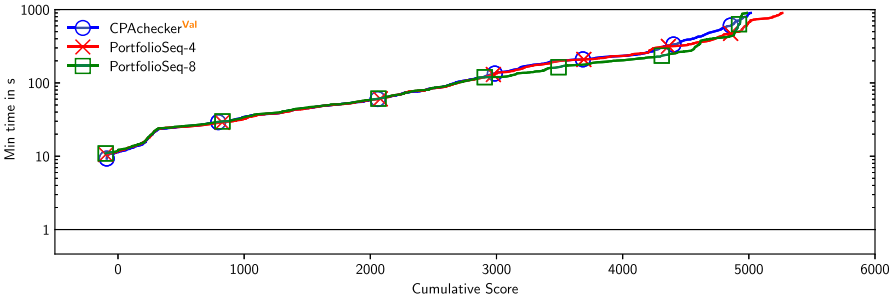

Fig. 10 Sequential portfolios: Score-based quantile plot comparing CPACHECKER^{Val}, the best and the worst-performing sequential portfolios (SeqPortfolio-4 and SeqPortfolio-8, respectively)

that sequential portfolios can be less resource-efficient in comparison to CPACHECKER^{Val}. In general, as we increase the size of the sequential portfolio and thereby increase its effectiveness, we also decrease its resource efficiency. The trend of increased effectiveness is visible for all sequential portfolios, except for the sequential portfolio of 8 tools. In this case, the verifiers in the portfolio of 8 tools are not provided with enough resources to solve some of

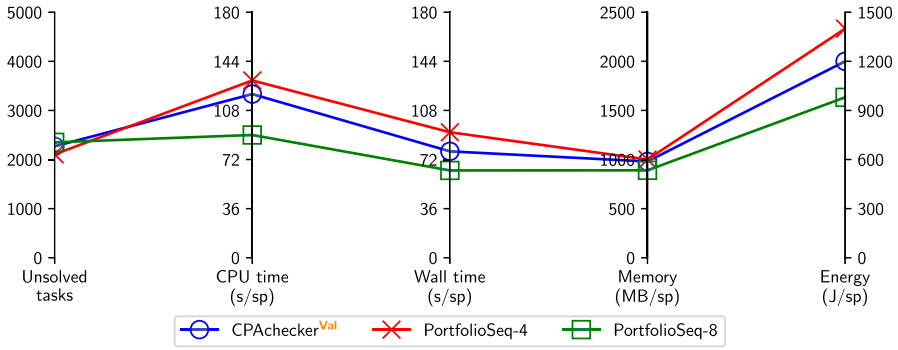

Fig. 11 Sequential portfolios: Unsolved tasks and resource consumption per score point of CPAchecker Val, the best and the worst-performing sequential portfolios (SeqPortfolio-4 and SeqPortfolio-8, respectively)

the given tasks, which reduces the effectiveness of the portfolio but also increases its resource efficiency for computing correct results.

This is also visible in the plot graph for the sequential portfolio of 8 tools in Fig. 10. Here the plot graph goes most toward the right in three steps and after that its slope increases considerably, showing that the later verifiers did not contribute much to the number of correctly solved tasks.

Difference to previous results [21]. A key observation of our previous work [21] is that portfolios are negatively affected by wrong results produced by unsound tools. As a consequence, sequential portfolios often perform worse overall even though they can achieve a higher number of correct results. In this work, we mitigate the impact of wrong results by employing validating verifiers. As a consequence, our sequential portfolios do not produce more incorrect results than the best standalone VERIFIER Val composition CPACHECKER Val. As a side effect, sequential portfolios are now able to achieve a higher score.

5.3 RQ 2: Evaluation of Parallel-Portfolio Verifiers

We now present the results for the parallel portfolio. Table 3 shows the summary of results for the parallel portfolios. Similar to the sequential portfolio, the parallel portfolio solves a higher number of verification tasks in comparison to CPACHECKER Val and thereby also achieves a higher score. Initially, the score increases with the size of a parallel portfolio. However, as the size becomes too large to give each verifier reasonable resources, the performance starts to decrease.

Figure 12 shows the quantile plot of normalized scores for the best and worst-performing parallel portfolios, and CPACHECKER Val. All graphs start from nearly the same abscissa because all of them have the very low number of incorrect results. The portfolio of size 4 goes farthest to the right because it has the highest score.

Figure 13 shows that the best-performing parallel portfolio performs better than CPACHECKER Val in terms of resource efficiency except for memory consumption. Higher memory consumption is expected as several tools are running in parallel. A lower wall-time is expected for the same reason. The reduction in CPU time is interesting, which we attribute to the diversity of the benchmark set: some tasks are simple for one tool but harder for another, and vice versa.

The resource consumption of the worst-performing parallel portfolio is worse than for CPACHECKER Val. Portfolios of large sizes do not provide enough resources for any verifier

Table 3 Parallel portfolios: Comparison of different size with CPAchecker

Verifier	CPAchecker^{Val}	Parallel Portfolio of			
		2	3	4	8
Score, normalized	5 016	5 300	5 309	**5 334**	4 849
Correct results	3 129	3 311	**3 384**	3 351	2 977
Correct proofs	1 574	**1 620**	1 596	1 614	1 322
Correct alarms	1 555	1 691	**1 788**	1 737	1 655
Wrong results	2	2	2	2	**1**
Wrong proofs	**0**	**0**	**0**	**0**	**0**
Wrong alarms	2	2	2	2	**1**
Total resource consumption for correct results					
CPU time(h)	170	**150**	160	180	230
Wall time (h)	110	56	43	36	**32**
Memory (GB)	**4 900**	7 200	8 400	9 500	11 000
CPU Energy (KJ)	6 200	4 500	4 100	**4 000**	**4 000**
Median resource consumption for correct results					
CPU time(s)	150	91	**83**	130	200
Wall time (s)	86	17	**16**	21	27
Memory (MB)	**1 100**	1 400	1 500	1 800	3 100
CPU Energy (J)	1 500	550	**530**	740	1 000
Resource consumption of correct results per score point					
CPU time (s/sp)	120	**100**	110	120	170
Wall time (s/sp)	78	38	29	**24**	**24**
Memory (MB/sp)	**980**	1 400	1 600	1 800	2 200
CPU Energy (J/sp)	1 200	840	770	**750**	830

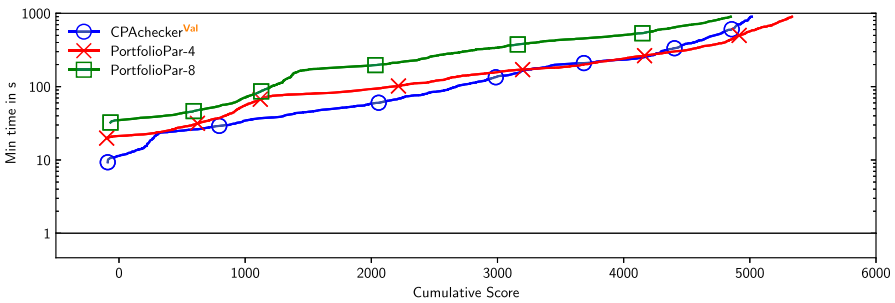

Fig. 12 Parallel portfolios: score-based quantile plot comparing CPAchecker^{Val}, the best and the worst-performing parallel portfolios (ParPortfolio-4 and ParPortfolio-8, respectively)

to compute a result. As a result, only those verification tasks get solved that are *easy* for at least one verifier in the portfolio. Nonetheless, since several tools are running in parallel, the CPU time and memory are still accounted for even this short execution time. We can see that the wall time for the parallel portfolio of size 8 is the least.

Difference to previous results [21]. In addition to producing a high number of wrong results, unsound tools often produce wrong results quickly [21]. For parallel portfolios, this impacts

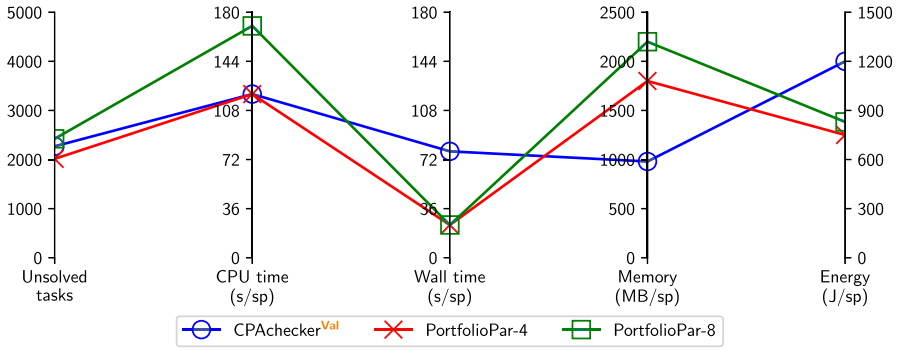

Fig. 13 Parallel portfolios: Unsolved tasks and resource consumption per score point of CPAchecker^Val, the best and the worst-performing parallel portfolios (ParPortfolio-4 and ParPortfolio-8, respectively)

the performance negatively as wrong results that are produced quickly are selected before a correct result can be computed. Therefore, similar to sequential portfolios, parallel portfolios based on validating verifiers did not produce a higher number of wrong results than the single best tool CPAchecker^Val. (Interestingly, the parallel portfolio of size 8 produces even fewer incorrect results. This was due to timeout.)

5.4 RQ 3: Evaluation of Algorithm-Selection Verifiers

Table 4 shows the summary of results for algorithm selection: There is a clear trend towards better results with more verifiers. This is expected because our selector has more options to choose from, including verifiers that are more effective for some tasks. Also, algorithm-selection-based verifiers do not need to share resources between verifiers. Therefore, they can benefit from multiple verifiers without wasting resources on unsuccessful verification attempts. The number of wrong results is, as expected, relatively low.

In Fig. 14, all plots start from around similar scores but at different times. Algorithm-selection-based verifiers have a higher startup time than the standalone CPAchecker^Val because of the overhead of the selection process. This difference in CPU time consumption is much more pronounced for the verification tasks that were solved quickly by the chosen verifier, but as the verifier starts dominating the CPU time consumption on more difficult tasks, this overhead of selection starts to pay off. We can observe that CPAchecker^Val performs initially better with respect to CPU time, but after around the midpoint, algorithm selection starts to be more efficient.

Figure 15 shows that algorithm selection is also more resource-efficient than CPAchecker^Val except for peak memory consumption. By design, the algorithm selector aims to predict the fastest verifier that solves the given task successfully.

There is a linear increase in CPU time and memory overhead with the number of choices the algorithm selector is given. We attribute this to using an off-the-shelf combination (see Fig. 4c) instead of an integrated one for the selection algorithm. Our construction allows adding a verifier just by adding the classifier based on hardness models. Increasing the number of tools for selection also increases the number of *classifiers* called. And since each of them is used as an off-the-shelf tool, the overhead of starting the classifier is added to the resource consumption. This explains the relatively high startup time for the algorithm selection with size 8. The quantile plot for peak memory consumption (Fig. 16) also shows

Table 4 Algorithm Selections: Comparison of different sizes with CPAchecker

Verifier	CPAchecker^{Val}	Algorithm Selection of			
		2	3	4	8
Score, normalized	5 016	5 268	5 420	5 563	**5 577**
Correct results	3 129	3 399	3 511	3 596	**3 604**
Correct proofs	1 574	1 632	1 647	1 716	**1 726**
Correct alarms	1 555	1 767	1 864	1 880	**1 878**
Wrong results	2	2	**1**	**1**	**1**
Wrong proofs	**0**	**0**	**0**	**0**	**0**
Wrong alarms	2	2	**1**	**1**	**1**
Total resource consumption for correct results					
CPU time(h)	170	**140**	**140**	150	170
Wall time (h)	110	94	92	92	**86**
Memory (GB)	4 900	**4 500**	4 600	5 000	6 000
CPU Energy (KJ)	6 200	**5 200**	**5 200**	5 500	5 400
Median resource consumption for correct results					
CPU time(s)	150	77	**74**	83	110
Wall time (s)	86	**30**	33	33	32
Memory (MB)	1 100	920	**850**	**850**	1 200
CPU Energy (J)	1 500	640	**630**	690	770
Resource consumption of correct results per score point					
CPU time (s/sp)	120	97	**94**	99	110
Wall time (s/sp)	78	64	61	60	**56**
Memory (MB/sp)	980	860	**840**	890	1 100
CPU Energy (J/sp)	1 200	1 000	**960**	980	980

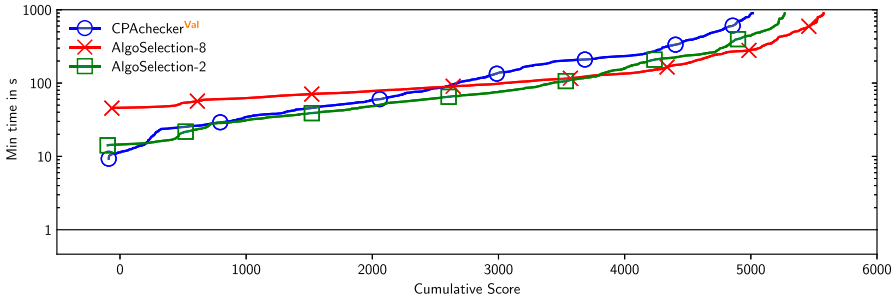

Fig. 14 Algorithm selections: Score-based quantile plot comparing CPAchecker^{Val}, the best and the worst performing algorithm selection based verifier^{Val} (AlgoSelection-8 and AlgoSelection-2, respectively)

this: it starts with higher memory consumption relative to CPAchecker^{Val} but then the line remains horizontal for most of the graph.[6]

[6] The version of CoVeriTeam used in our previous work [21] did not execute the classifiers concurrently so the peak memory consumption was not high. The (conceptually) parallel composition was implemented by executing the tools one after another and then combining the results. The newer version of CoVeriTeam executes tools concurrently. Due to this reason, we did not notice the increase in memory consumption in the results for algorithm selection of size 8 in our previous work [21].

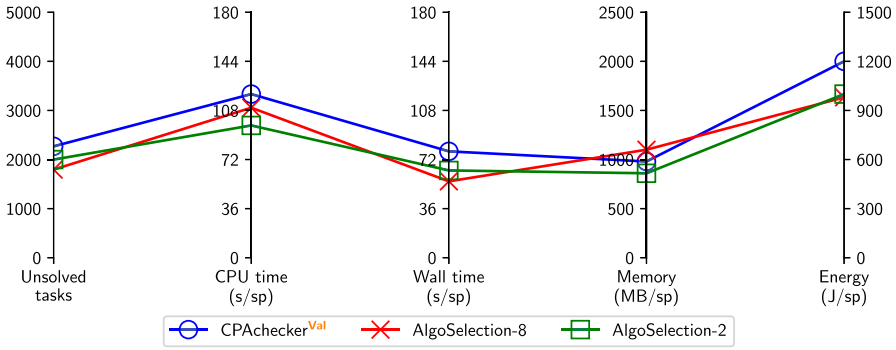

Fig. 15 Algorithm selections: Unsolved tasks and resource consumption per score point CPACHECKER^{Val}, the best and the worst performing algorithm selection based VERIFIER^{Val} (AlgoSelection-8 and AlgoSelection-2, respectively)

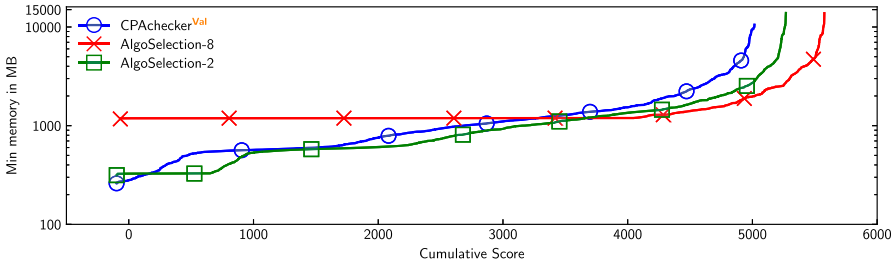

Fig. 16 Algorithm selections: Score-based quantile plot for memory consumption comparing CPACHECKER^{Val}, the best and the worst performing algorithm selection based VERIFIER^{Val} (AlgoSelection-8 and AlgoSelection-2, respectively)

Difference to Previous Results. **[21]** . The main results for the algorithm-selection-based verifiers confirm the results presented in our previous work [21]. There is one improvement though: In our previous work, we employed a simpler algorithm selector that did not consider the resource consumption of individual verifiers during selection. In contrast, our new algorithm selector prioritizes verifiers that solve a verification task not only correctly but also quickly. This change in design is visible in our experimental results. The algorithm-selection-based verifiers consume significantly fewer resources, which is visible for both CPU and wall time.

5.5 Discussion

Our experiments show that each combination can on average perform better than any standalone VERIFIER^{Val} in terms of correctly solved tasks. This is also true for the normalized scores.

We were expecting that portfolios would be less effective in comparison to the standalone tools because of higher resource consumption. In particular, we were expecting that they would be unable to solve hard tasks as less resources would be allocated to each participating tool. However, the experimental data demonstrate the opposite. A portfolio would be unable to solve tasks that are hard for each tool in the portfolio. Our benchmark set had few such tasks. But for most of the tasks that were hard for one tool, there was some

other tool in the portfolio that could solve it in the allocated time. This was especially pronounced in the parallel portfolio.

The outcome regarding resource consumption is in agreement with our expectations. In comparison to the best performing standalone VERIFIER[Val], sequential portfolios require more time but perform better with respect to memory consumption. Whereas, parallel portfolios perform better with respect to wall-time but have higher memory consumption. It seems that the portfolios are more energy-efficient when more cores are being used.

Our algorithm selection is based on a model trained using machine learning. The training penalizes the tools that produce more incorrect results and also considers the resource consumption in terms of CPU time. In comparison to both portfolios, the verifier based on algorithm selection solved more tasks.

Our verifier combinations can be constructed by simply selecting tools that perform well in a comparative evaluation, such as the Competition on Software Verification. We found that it leads to successful combinations for all evaluated combination types. Nevertheless, the combinations can be further fine-tuned to achieve even better results.

The portfolio combinations are easy to construct and can perform well if the set of tools to combine is diverse (different strengths). Also, the portfolios should not be too large unless we are willing to increase the resources. Training the algorithm selector requires more preliminary work, but with limited resources and enough choices (number of tools), the selection-based verifier becomes more effective.

Difference to Previous Results [21]. A key observation of our previous work [21] is that portfolios prefer fast results and unsound tools may produce wrong results quickly. Therefore, the soundness of portfolios can be affected by fast unsound tools. Since this could partially be addressed by the execution order in sequential portfolios, this effect was more pronounced in parallel portfolios than in sequential portfolios. To mitigate this problem in general, we suggested to introduce a validation step. In this work, we adopted the proposed mitigation strategy and updated our experiment setup to include validation. Our results show that it achieves the intended effect as the number of incorrect results decreases significantly.

Correct vs. Wrong Results. Interestingly, and in line with our previous work [21], we can observe a tradeoff between correct results and wrong results. By combining tools, we can easily increase the number of solved tasks but we also risk increasing the number of wrong results. The validation step introduced in this work effectively reduces the number of wrong results but also limits the number of tasks that can be correctly verified by the individual verifiers.

In the end, our works present three types of easy-to-deploy combinations of software verifiers, both with and without validation. A user can choose to include validation if the user prefers to ensure correct results, otherwise, the user can choose to omit the validation step. Furthermore, we are convinced that investigating combination types with a better tradeoff of correct vs. wrong results is an exciting research direction.

Type of Verification Tools. CoVeriTeam supports only automated verification tools, therefore, all the tools in a combination must be automated. For our combinations, we have considered tools based on their performance in SV-COMP. These happen to be model checkers, instead of deductive software verifiers like Dafny [60], Frama-C [61], KeY [62], VerCors [63], VeriFast [64], etc.

Any automated verifier that runs on a Linux system can be integrated in CoVeriTeam. To integrate a new tool, one needs to create a self-contained archive that is available online, write a tool-info module to assemble the command line and process the output, an actor definition for CoVeriTeam to orchestrate the execution of the tool and process artifacts, and have a corresponding actor in CoVeriTeam. CoVeriTeam already contains a library of

verification actors and actor definitions for many publicly available verification tools. More details on how to integrate a new tool are available in the article on CoVeriTeam [18].

Comparison Between Different Types of Combinations. Our evaluation compares different types of combinations with the best-performing standalone tool. Although it is enticing, interesting, and valuable to investigate how different combinations perform in comparison to each other, this paper focuses only on how they compare to standalone tools.

A comparative evaluation of different combinations would require a different experiment setup. Different factors influence the performance of each of these combinations, e.g., the position of a verifier in a sequential portfolio, the training set for the learning-based selector, set of chosen verifiers in portfolios. Our experiment setup uses the top n verifiers based on a list and used them in these combinations. This setup is inadequate to conclude how these combinations would perform against each other. We leave the optimal configuration of our combination types and the comparison between combinations open for future work.

6 Threats to Validity

6.1 External Validity

Selection of verifiers. The effectivity of a combination of tools depends on the effectivity of its parts. Therefore, the performance of a concrete instantiation of our tool combinations is influenced by the selected tools and their configuration, and our results might not generalize to other selections of tools. We have selected the eight most powerful verification tools of the category ReachSafety based on the results of the competition SV-COMP and executed them in their original configuration as submitted to the competition. Our procedure to select the verifiers to include in the combination is described in Sect. 4.1.

Applicability to other verification tasks. Our evaluation results are based on experiments with a given benchmark set. While we have evaluated our tool combinations on programs taken from the largest and most diverse set of publicly available verification tasks for C programs, the performance of the evaluated combinations may be different on other sets of verification tasks. The selection of the benchmark set is described in Sect. 4.7.

Training of the algorithm selector The choice of the benchmark set also impacts the training of our algorithm selector. Training a learning-based selector requires a large and diverse set of verification tasks. Each task has to be labeled with the execution results of each verifier used in our combinations. The used benchmarks repository [56, 57] was created and improved by the verification community over many years. We are not aware of any other benchmark set of verification tasks that is as diverse and of the same quality as this one. As a result, we had to train our algorithm selector on the same dataset that we later used for benchmarking the tool combinations. Therefore, our evaluation shows that algorithm selection improves the performance of verification on the given benchmark set and the selector might generalize only to benchmark sets with similarly distributed verification tasks. For a fair comparison, we (1) restricted the training to linear models, which are known to generalize well, (2) trained only on a random subset of the benchmark set, 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.

Design of the algorithm selector The evaluation of algorithm selection is also dependent on the chosen selection technique. Choosing alternative selection methods, e.g., based on hand-

crafted rules, might impact the evaluation. However, the design of hand-crafted methods is not straightforward, it might require expert knowledge about the tool implementations of the components. This design process might in addition be biased in favor of certain tool combinations, which could also impact the experimental results.

Experiment environment. The setup of our experiments is influenced by SV-COMP: benchmark set, tool selection, the configuration in which tools are used, execution environment, and resource limitations. On the one hand, it gives us the benefit that our results could be compared with the evaluation of many publicly available well-known verification tools, on the other hand, it affects the generalizability of our results. However, over the last decade, the setup used by SV-COMP has become standard in the verification community for the evaluation of verification tools and this was the best choice available to us. Also, using the SV-COMP setup allows us to compare the results of our combinations with the results of the standalone tools from SV-COMP 2022.

Sequential portfolios. The order of the verifiers in sequential portfolios may impact its performance. We ordered the verifiers in sequence according to their performance in SV-COMP 2022, that is, the best performing tool is executed first, and so on. Changing the order of the tools might change the results concerning resource consumption. We noted in our previous work [21] that changing the order of verifiers can impact the soundness of the combination. This was happening if an unsound and fast verifier was put early in the sequence. We now mitigate this issue by validating the results produced by the verifiers.

Validation step. The results of our evaluation are dependent on the quality of employed validators and witnesses produced by the verifiers. We have used the validators participating in SV-COMP 2022. The selection of the validators to include in the combination is described in Sect. 4.2. We used a portfolio of validators because no standalone validator could effectively validate the results produced by most of the verifiers. Each validator was more effective for some subset of verifiers. Our results could change with a different selection of validators or a different quality of witnesses produced by the verifiers.

6.2 Internal Validity

Experiment setup. We have used the same verifier archives, benchmark set, benchmarking framework, resource limits, and infrastructure to execute our experiments as was used for SV-COMP 2022. The unchanged execution setup ensures that there are no unintended side effects in our experiments. Also, since we did not change any component of the verifiers and executed them with the same parameters inside the combination and when executed as a standalone verifier, we exclude the possibility that we could have used the verifiers in a sub-optimal way.

Memory and time overhead. CoVeriTeam induces an overhead of about 0.8 s for each actor in the combination and around 44 MB memory overhead [18]. It is possible to reduce this overhead by using shell scripts, but we decided in favor of using CoVeriTeam for composing tools because it supports modular design. This is especially pronounced in our algorithm-selector combination. We could have saved a few seconds if we were using a monolithic algorithm selector instead of composing one.

Measurement and control of resources. We have used BenchExec [37] to measure CPU time and memory consumption, and to enforce the resource limits. Since BenchExec is based on the modern features of the Linux kernel and thus the most accurate measurement technology, we eliminate the measurement-related confounding factors in our evaluation according to the state of the art.

Distribution of CPU cores. We rely on the operating system for a fair distribution of CPU usage during the execution of parallel portfolios. In general, there might be a tool that uses multi-processing and as a result would consume more CPU time as compared to a tool using only one process. In such a portfolio, where some tools heavily use multi-processing and others use only one process, the actors using multi-processing would unfairly consume more CPU time and deliver more results. CoVeriTeam can only control the limits. A user of a portfolio can limit the CPU time available to each tool, resulting in the termination of tools that consumed the allocated CPU time, but whether CPU time is consumed sooner or later by a process is decided by the operating system. In our experiments with parallel portfolios, we allow tools to use the CPU time left by tools that terminated earlier.

6.3 Construct Validity

Our experiments are designed to assess whether combinations of verifiers can improve effectivity and efficiency compared to standalone verifiers. The measures that we use to quantify the quality are the community-agreed scoring schema, the number of solved tasks, and the resource measures memory, CPU time, wall time, and CPU energy. These measures are all standards accepted by the verification community and have also been used in the competition on software verification for many years.

7 Related work

Combination Types Used in Software Verification. Combining verifiers to increase the verification performance is a well-established technique in the domain of software verification [14, 23, 24, 27, 29, 50, 51, 65–68]. The top three winning entries of the software-verification competition SV-COMP 2022 all combine various verification techniques to achieve their performance [5]. CPAchecker [29] combines up to six different verification approaches into three sequential portfolios that are task-dependently selected with an algorithm selector. VeriAbs [49] employs up to nine different verification approaches that are combined into four verification strategies and task-dependently selected by an algorithm selector. PeSCo [50] ranks individual verification algorithms according to their predicted likelihood of solving a given task and then executes them sequentially in descending order. PredatorHP [65] and Ufo [66] 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 [67] distribute the workload of a single verification task among multiple algorithms to combine their strengths. For example, conditional model checking [69–72] runs two or more verifiers in sequence, while the program is reduced after every step to the state space of the program that is left unexplored by the previous algorithm. CoVeriTest [73, 74], a tool for test-case generation based on verification, interleaves multiple verifiers, while (partially) sharing the analysis state between algorithms. MetaVal [55] integrates verification tools for witness validation (i.e., to check whether a verifier had produced a valid 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 the tool level requires the exchange of analysis information between tools. In general, existing verification tools are not well suited for this type of combination, which leads us to explore off-the-shelf 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 has been successfully applied in many research areas including SAT solving [75–77], constraint-satisfaction programs [78–80], and combinatorial-search problems [81]. The employed approaches traditionally focused on portfolio-based approaches [75, 76, 79], but recent techniques started to integrate algorithm selectors for either selecting single algorithms [77, 78] or portfolios of algorithms [80, 82]. For example, earlier works in SAT solving [75, 76] focused on parallel-portfolio solvers, while later works such as SATZILLA [77] further improve 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 [83] 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 can yield performance gains.

8 Conclusion

This paper describes a method to construct combinations of verification and validation tools in a systematic and modular way. The method does not require any changes to the tools that are used to construct the combinations. Given the large number of freely available verifiers and validators for C programs, there is a huge potential for improvement in effectivity and efficiency (a total of 50 verifiers and 13 validators were evaluated in SV-COMP 2024 [84]). 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. Our contribution is to offer an easy way for practitioners to benefit from the available verification tools and leverage better performance from the latest research and development efforts in software verification.

Funding Open Access funding enabled and organized by Projekt DEAL. This work was funded in part by Deutsche Forschungsgesellschaft (DFG)—378803395 (ConVeY) and 418257054 (Coop).

Data availability A reproduction package including all our results is available at Zenodo [22]. Additionally, the result tables are also available on a supplementary web page for convenient browsing: https://www.sosy-lab.org/research/coveriteam-combinations.

References

1. Beyer D, Podelski A (2022) Software model checking: 20 years and beyond. In: Principles of systems design. LNCS, vol 13660. Springer, pp 554–582. https://doi.org/10.1007/978-3-031-22337-2_27
2. Hoare CAR (2003) The verifying compiler: a grand challenge for computing research. J. ACM 50(1):63–69. https://doi.org/10.1145/602382.602403
3. Clarke EM, Henzinger TA, Veith H, Bloem R (2018) Handbook of model checking. Springer, Berlin. https://doi.org/10.1007/978-3-319-10575-8
4. Jhala R, Majumdar R (2009) Software model checking. ACM Comput Surv. doi 10(1145/1592434):1592438
5. Beyer D (2022) Progress on software verification: SV-COMP 2022. In: Proceedings of TACAS (2), LNCS, vol 13244. Springer, pp 375–402. https://doi.org/10.1007/978-3-030-99527-0_20
6. Beckert B, Hähnle R (2014) Reasoning and verification: state of the art and current trends. IEEE Intell Syst 29(1):20–29. https://doi.org/10.1109/MIS.2014.3
7. Beyer D, Gulwani S, Schmidt D (2018) Combining model checking and data-flow analysis. In: Handbook of model checking. Springer, pp 493–540. https://doi.org/10.1007/978-3-319-10575-8_16
8. Garavel H, ter Beek MH, van de Pol J (2020) The 2020 expert survey on formal methods. In: Proceedings of FMICS. LNCS, vol 12327. Springer, pp 3–69.https://doi.org/10.1007/978-3-030-58298-2_1
9. Ball T, Rajamani SK (2002) The SLAM project: debugging system software via static analysis. In: Proceedings of POPL. ACM, pp 1–3.https://doi.org/10.1145/503272.503274
10. Khoroshilov AV, Mutilin VS, Petrenko AK, Zakharov V (2009) Establishing Linux driver verification process. In: Proceedings of Ershov memorial conference. LNCS, vol 5947. Springer, pp 165–176. https://doi.org/10.1007/978-3-642-11486-1_14
11. Chong N, Cook B, Eidelman J, Kallas K, Khazem K, Monteiro FR, Schwartz-Narbonne D, Tasiran S, Tautschnig M, Tuttle MR (2021) Code-level model checking in the software development workflow at Amazon Web Services. Softw Pract Exp 51(4):772–797. https://doi.org/10.1002/spe.2949
12. Calcagno C, Distefano D, Dubreil J, Gabi D, Hooimeijer P, Luca M, O'Hearn PW, Papakonstantinou I, Purbrick J, Rodriguez D (2015) Moving fast with software verification. In: Proceedings of NFM. LNCS, vol 9058. Springer, pp 3–11. https://doi.org/10.1007/978-3-319-17524-9_1
13. Beyer D, Keremoglu ME (2011) CPACHECKER: a tool for configurable software verification. In: Proceedings of CAV. LNCS, vol 6806. Springer, pp 184–190.https://doi.org/10.1007/978-3-642-22110-1_16
14. Dangl M, Löwe S, Wendler P (2015) CPACHECKER with support for recursive programs and floating-point arithmetic (competition contribution). In: Proceedings of TACAS. LNCS, vol 9035. Springer, pp 423–425.https://doi.org/10.1007/978-3-662-46681-0_34
15. Gadelha MYR, Monteiro FR, Cordeiro LC, Nicole DA (2019) ESBMC v6.0: verifying C programs using k-induction and invariant inference (competition contribution). In: Proceedings of TACAS (3). LNCS, vol 11429. Springer, pp 209–213.https://doi.org/10.1007/978-3-030-17502-3_15
16. Huberman BA, Lukose RM, Hogg T (1997) An economics approach to hard computational problems. Science 275(7):51–54. https://doi.org/10.1126/science.275.5296.51
17. Rice JR (1976) The algorithm selection problem. Adv Comput 15:65–118. https://doi.org/10.1016/S0065-2458(08)60520-3
18. Beyer D, Kanav S (2022) COVERITEAM: on-demand composition of cooperative verification systems. In: Proceedings of TACAS. LNCS, vol 13243. Springer, pp 561–579. https://doi.org/10.1007/978-3-030-99524-9_31
19. Beyer D, Kanav S, Wachowitz H. Source-code repository of COVERITEAM. https://gitlab.com/sosy-lab/software/coveriteam. Accessed 09 Feb 2023
20. Beyer D, Kanav S, Wachowitz H (2023) COVERITEAM service: verification as a service. In: Proceedings of ICSE, companion. IEEE, pp 21–25.https://doi.org/10.1109/ICSE-Companion58688.2023.00017
21. Beyer D, Kanav S, Richter C (2022) Construction of verifier combinations based on off-the-shelf verifiers. In: Proceedings of FASE. Springer, pp 49–70. https://doi.org/10.1007/978-3-030-99429-7_3
22. Beyer D, Kanav S, Kleinert T, Richter C (2023) Reproduction package for FMSD article 'Construction of verifier combinations from off-the-shelf components'. Zenodo. https://doi.org/10.5281/zenodo.7838348
23. Wendler P (2013) CPACHECKER with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In: Proceedings of TACAS. LNCS, vol 7795. Springer, pp 613–615. https://doi.org/10.1007/978-3-642-36742-7_45
24. Heizmann M, Chen YF, Dietsch D, Greitschus M, Hoenicke J, Li Y, Nutz A, Musa B, Schilling C, Schindler T, Podelski A (2018) ULTIMATE AUTOMIZER and the search for perfect interpolants (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 10806. Springer, pp 447–451. https://doi.org/10.1007/978-3-319-89963-3_30

25. Kotoun M, Peringer P, Šoková V, Vojnar T (2016) Optimized PredatorHP and the SV-COMP heap and memory safety benchmark (competition contribution). In: Proceedings of TACAS. LNCS, vol 9636. Springer, pp 942–945.https://doi.org/10.1007/978-3-662-49674-9_66
26. Holík L, Kotoun M, Peringer P, Šoková V, Trtík M, Vojnar T (2016) Predator shape analysis tool suite. Proceedings of HVC. LNCS 10028:202–209. https://doi.org/10.1007/978-3-319-49052-6_13
27. Richter C, Hüllermeier E, Jakobs MC, Wehrheim H (2020) Algorithm selection for software validation based on graph kernels. Autom Softw Eng 27(1):153–186. https://doi.org/10.1007/s10515-020-00270-x
28. Richter C, Wehrheim H (2020) Attend and represent: a novel view on algorithm selection for software verification. In: Proceedings of ASE, pp 1016–1028. https://doi.org/10.1145/3324884.3416633
29. Beyer D, Dangl M (2018) Strategy selection for software verification based on boolean features: a simple but effective approach. In: Proceedings of ISoLA. LNCS, vol 11245. Springer, pp 144–159. https://doi.org/10.1007/978-3-030-03421-4_11
30. Demyanova Y, Pani T, Veith H, Zuleger F (2017) Empirical software metrics for benchmarking of verification tools. Formal Methods Syst Des 50(2–3):289–316. https://doi.org/10.1007/s10703-016-0264-5
31. Beyer D, Dangl M, Dietsch D, Heizmann M, Lemberger T, Tautschnig M (2022) Verification witnesses. ACM Trans Softw Eng Methodol 31(4):57:1-57:69. https://doi.org/10.1145/3477579
32. Beyer D, Dangl M, Dietsch D, Heizmann M, Stahlbauer A (2015) Witness validation and stepwise testification across software verifiers. In: Proceedings of FSE. ACM, pp 721–733. https://doi.org/10.1145/2786805.2786867
33. Beyer D, Dangl M, Dietsch D, Heizmann M (2016) Correctness witnesses: exchanging verification results between verifiers. In: Proceedings of FSE. ACM, pp 326–337. https://doi.org/10.1145/2950290.2950351
34. Beyer D, Dangl M, Lemberger T, Tautschnig M (2018) Tests from witnesses: execution-based validation of verification results. In: Proceedings of TAP. LNCS, vol 10889. Springer, pp 3–23. https://doi.org/10.1007/978-3-319-92994-1_1
35. Beyer D, Dangl M (2016) Verification-aided debugging: an interactive web-service for exploring error witnesses. In: Proceedings of CAV (2). LNCS, vol 9780. Springer, pp 502–509.https://doi.org/10.1007/978-3-319-41540-6_28
36. Beyer D (2015) Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proceedings of TACAS. LNCS, vol 9035. Springer, pp 401–416.https://doi.org/10.1007/978-3-662-46681-0_31
37. Beyer D, Löwe S, Wendler P (2019) Reliable benchmarking: requirements and solutions. Int J Softw Tools Technol Transf 21(1):1–29. https://doi.org/10.1007/s10009-017-0469-y
38. Kleinert T (2022) Developing a verifier based on parallel portfolio with CoVeriTeam. Bachelor's Thesis, LMU Munich, Software Systems Lab
39. Demyanova Y, Veith H, Zuleger F (2013) On the concept of variable roles and its use in software analysis. In: Proceedings of FMCAD. IEEE, pp 226–230. https://doi.org/10.1109/FMCAD.2013.6679414
40. Apel S, Beyer D, Friedberger K, Raimondi F, Rhein A (2013) Domain types: abstract-domain selection based on variable usage. In: Proceedings of HVC. LNCS, vol 8244. Springer, pp 262–278. https://doi.org/10.1007/978-3-319-03077-7
41. Xu L, Hoos HH, Leyton-Brown K (2007) Hierarchical hardness models for SAT. In: International conference on principles and practice of constraint programming. Springer, pp 696–711. https://doi.org/10.1007/978-3-540-74970-7_49
42. Kadioglu S, Malitsky Y, Sabharwal A, Samulowitz H, Sellmann M (2011) Algorithm selection and scheduling. In: Proceedings of CP. Springer, pp 454–469.https://doi.org/10.1007/978-3-642-23786-7_35
43. Torgo L, Gama J (1996) Regression by classification. In: Brazilian symposium on artificial intelligence. Springer, pp 51–60.https://doi.org/10.1007/3-540-61859-7_6
44. Chalupa M, Řechtáčková A, Mihalkovič V, Zaoral L, Strejček J (2022) Symbiotic 9: string analysis and backward symbolic execution with loop folding (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 13244. Springer, pp 462–467.https://doi.org/10.1007/978-3-030-99527-0_32
45. Kröning D, Tautschnig M (2014) Cbmc: C bounded model checker (competition contribution). In: Proceedings of TACAS. LNCS, vol 8413. Springer, pp 389–391.https://doi.org/10.1007/978-3-642-54862-8_26
46. Chaudhary E, Joshi S (2019) Pinaka: symbolic execution meets incremental solving (competition contribution). In: Proceedings of TACAS (3). LNCS, vol 11429. Springer, pp 234–238. https://doi.org/10.1007/978-3-030-17502-3_20
47. Dietsch D, Heizmann M, Nutz A, Schätzle C, Schüssele F (2020) Ultimate Taipan with symbolic interpretation and fluid abstractions (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 12079. Springer, pp 418–422.https://doi.org/10.1007/978-3-030-45237-7_32
48. Malík V, Schrammel P, Vojnar T (2020) 2ls: Heap analysis and memory safety (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 12079. Springer, pp 368–372. https://doi.org/10.1007/978-3-030-45237-7_22

49. Darke P, Agrawal S, Venkatesh R (2021) VeriAbs: a tool for scalable verification by abstraction (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 12652. Springer, pp 458–462.https://doi.org/10.1007/978-3-030-72013-1_32
50. Richter C, Wehrheim H (2019) PeSCo: predicting sequential combinations of verifiers (competition contribution). In: Proceedings of TACAS (3). LNCS, vol 11429. Springer, pp 229–233.https://doi.org/10.1007/978-3-030-17502-3_19
51. Leeson W, Dwyer, M (2022) Graves-CPA: a graph-attention verifier selector (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 13244. Springer, pp 440–445. https://doi.org/10.1007/978-3-030-99527-0_28
52. Beyer D, Strejček J (2022) Case study on verification-witness validators: where we are and where we go. In: Proceedings of SAS. LNCS, vol 13790. Springer, pp 160–174. https://doi.org/10.1007/978-3-031-22308-2_8
53. Ayaziová P, Chalupa M, Strejček J (2022) Symbiotic-Witch: a Klee-based violation witness checker (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 13244. Springer, pp 468–473.https://doi.org/10.1007/978-3-030-99527-0_33
54. Švejda J, Berger P, Katoen JP (2020) Interpretation-based violation witness validation for C: NitWit. In: Proceedings of TACAS. LNCS, vol 12078. Springer, pp 40–57. https://doi.org/10.1007/978-3-030-45190-5_3
55. Beyer D, Spiessl M (2020) MetaVal: witness validation via verification. In: Proceedings of CAV. LNCS, vol 12225. Springer, pp 165–177.https://doi.org/10.1007/978-3-030-53291-8_10
56. Collection of verification tasks. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks. Accessed 01 Apr 2023
57. Beyer D (2022) SV-benchmarks: benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo. https://doi.org/10.5281/zenodo.5831003
58. Beyer D (2013) Second competition on software verification (Summary of SV-COMP 2013). In: Proceedings of TACAS. LNCS, vol 7795. Springer, pp 594–609. https://doi.org/10.1007/978-3-642-36742-7_43
59. Beyer D (2012) Competition on software verification (SV-COMP). In: Proceedings of TACAS. LNCS, vol 7214. Springer, pp 504–524.https://doi.org/10.1007/978-3-642-28756-5_38
60. Leino KRM (2010) Dafny: an automatic program verifier for functional correctness. In: Proceedings of LPAR. LNCS, vol 6355. Springer, pp 348–370.https://doi.org/10.1007/978-3-642-17511-4_20
61. Cuoq P, Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B (2012) Frama-C. In: Proceedings of SEFM. Springer, pp 233–247.https://doi.org/10.1007/978-3-642-33826-7_16
62. Ahrendt W, Baar T, Beckert B, Bubel R, Giese M, Hähnle R, Menzel W, Mostowski W, Roth A, Schlager S, Schmitt PH (2005) The key tool. Softw Syst Model 4(1):32–54. https://doi.org/10.1007/s10270-004-0058-x
63. Blom S, Huisman M (2014) The VerCors tool for verification of concurrent programs. In: Proceedings of FM. LNCS, vol 8442. Springer, pp 127–131. https://doi.org/10.1007/978-3-319-06410-9_9
64. Jacobs B, Smans J, Philippaerts P, Vogels F, Penninckx W, Piessens F (2011) VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Proceedings of NFM. LNCS, vol 6617. Springer, pp 41–55.https://doi.org/10.1007/978-3-642-20398-5_4
65. Peringer P, Šoková V, Vojnar T (2020) PredatorHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 12079. Springer, pp 408–412.https://doi.org/10.1007/978-3-030-45237-7_30
66. Albarghouthi A, Li Y, Gurfinkel A, Chechik M (2012) Ufo: a framework for abstraction- and interpolation-based software verification. In: Proceedings of CAV. LNCS, vol 7358. Springer, pp 672–678. https://doi.org/10.1007/978-3-642-31424-7_48
67. Beyer D, Wehrheim H (2020) Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proceedings of ISoLA (1). LNCS, vol 12476. Springer, pp 143–167.https://doi.org/10.1007/978-3-030-61362-4_8
68. Filliâtre JC, Paskevich A (2013) Why3: where programs meet provers. In: Programming languages and systems. Springer, pp 125–128.https://doi.org/10.1007/978-3-642-37036-6_8
69. Beyer D, Henzinger TA, Keremoglu ME, Wendler P (2012) Conditional model checking: a technique to pass information between verifiers. In: Proceedings of FSE. ACM. https://doi.org/10.1145/2393596.2393664
70. Beyer D, Jakobs MC, Lemberger T, Wehrheim H (2018) Reducer-based construction of conditional verifiers. In: Proceedings of ICSE. ACM, pp 1182–1193. https://doi.org/10.1145/3180155.3180259
71. Beyer D, Jakobs MC (2020) Fred: conditional model checking via reducers and folders. In: Proceedings of SEFM. LNCS, vol 12310. Springer, pp 113–132. https://doi.org/10.1007/978-3-030-58768-0_7
72. Beyer D, Jakobs MC, Lemberger T (2020) Difference verification with conditions. In: Proceedings of SEFM. LNCS, vol 12310. Springer, pp 133–154.https://doi.org/10.1007/978-3-030-58768-0_8

73. Beyer D, Jakobs MC (2019) CoVeriTest: cooperative verifier-based testing. In: Proceedings of FASE. LNCS, vol 11424. Springer, pp 389–408.https://doi.org/10.1007/978-3-030-16722-6_23
74. Beyer D, Jakobs MC (2021) Cooperative verifier-based testing with CoVeriTest. Int J Softw Tools Technol Transf 23(3):313–333. https://doi.org/10.1007/s10009-020-00587-8
75. Wotzlaw A, van der Grinten A, Speckenmeyer E, Porschen S (2012) pfolioUZK: solver description. In: Proceedings of SAT challenge p 45. https://helda.helsinki.fi/handle/10138/34218
76. Roussel O (2011) Description of portfolio. In: Proceedings of SAT challenge, p 46. https://helda.helsinki.fi/handle/10138/34218
77. Xu L, Hutter F, Hoos HH, Leyton-Brown K (2008) SATzilla: Portfolio-based algorithm selection for SAT. JAIR 32:565–606. https://doi.org/10.1613/jair.2490
78. Minton S (1996) Automatically configuring constraint satisfaction programs: a case study. Constraints 1(1–2):7–43. https://doi.org/10.1007/BF00143877
79. Bordeaux L, Hamadi Y, Samulowitz H (2009) Experiments with massively parallel constraint solving. In: Proceedings of IJCAI. https://www.ijcai.org/Proceedings/09/Papers/081.pdf
80. Yun X, Epstein SL (2012) Learning algorithm portfolios for parallel execution. In: Proceedings of LION. Springer, pp 323–338. https://doi.org/10.1007/978-3-642-34413-8_23
81. Kotthoff L (2016) Algorithm selection for combinatorial search problems: a survey. In: Data mining and constraint programming—foundations of a cross-disciplinary approach. LNCS, vol 10101. Springer, pp 149–190.https://doi.org/10.1007/978-3-319-50137-6_7
82. Lindauer M, Hoos H, Hutter F (2015) From sequential algorithm selection to parallel portfolio selection. In: Proceedings of LION. Springer, pp 1–16.https://doi.org/10.1007/978-3-319-19084-6_1
83. Kashgarani H, Kotthoff L (2021) Is algorithm selection worth it? Comparing selecting single algorithms and parallel execution. In: AAAI workshop on meta-learning and MetaDL challenge. PMLR, pp 58–64. https://proceedings.mlr.press/v140/kashgarani21a.html
84. Beyer D (2024) State of the art in software verification and witness validation: SV-COMP 2024. In: Proceedings of TACAS (3). LNCS, vol 14572. Springer, pp 299–329. https://doi.org/10.1007/978-3-031-57256-2_15