Revisiting Precision Reuse – A Reproduction and Replication Study

Marie-Christine Jakobs
Department of Computer Science
LMU Munich
Munich, Germany
m.jakobs@lmu.de

Márk Somorjai
Department of Computer Science
LMU Munich
Munich, Germany
mark.somorjai@lmu.de

***Abstract*—Automatic, formal software verification techniques are valuable means for software quality assurance because we can use them to show that a program fulfills certain correctness criteria. However, the program changes made during program evolution may invalidate verification results. Hence, we need to (re)verify every program version. Unfortunately, a complete verification of every program version is too costly. In 2013, precision reuse was introduced as a technique to efficiently reverify modified programs with abstraction-based verifiers, a well-established class of automatic software verifiers that iteratively analyze less and less abstract models of a program until the program is proven correct or a definite violation is detected in the program. Despite its broad applicability, its promising results in context of the verification tool CPACHECKER in 2013, and the need for efficient verification approaches, precision reuse has hardly been adopted by existing abstraction-based verifiers. Before we promote precision reuse, we aim to investigate the value of precision reuse in 2025. In particular, we aim to examine whether (a) significant advances in verification technology may lessen the benefits of precision reuse observed in 2013 and (b) whether the benefits of precision reuse generalize, e.g., carry over to other verifiers. To answer these questions, we perform an external reproduction and replication study with a recent CPACHECKER version, the additional verifier THETA, and an extended regression benchmark suite. Our study shows that the major claims and observations mostly remain valid and carry over but the benefits of precision reuse are less significant.**

***Index Terms*—model checking, program verification, specifying and verifying and reasoning about programs.**

I. INTRODUCTION

Automatic, formal software verification [1], [2] allows us to check whether a program violates or fulfills certain correctness criteria. To this end, abstraction-based verifiers [2] like e.g., SLAM [3], BLAST [4], CPACHECKER [5], [6], THETA [7], SATABS [8], [9], and ULTIMATEAUTOMIZER [10], analyze abstract models of the program. Depending on the analysis type, an abstract model may for instance only track the values of a subset of the program variables or abstract data states using a set of predicates over program variables. In general, one can configure the level of abstraction by fixing the information (e.g., tracked variables or predicates) tracked in the abstract model in general or at particular program locations. We refer to the information that fixes the abstraction level as the (abstraction) *precision*. To compute an appropriate abstraction precision, abstraction-based verifiers typically follow the iterative verify-refine approach sketched in Fig. 1. Starting with an initial, often coarse abstraction precision π_0, each iteration first analyzes the program on the abstraction level defined by the current abstraction precision π. If the abstraction is sufficient to prove the analyzed correctness criteria or to detect a real violation in the program, the iterative procedure will stop with the respective verification result. Otherwise, the procedure employs counterexample-guided abstraction refinement (CEGAR) [11] to refine the current abstraction precision using the detected spurious counterexample (i.e., a violation of the correctness criteria in the current abstract model), which does not exist in the program itself, and then, continues with the next iteration.

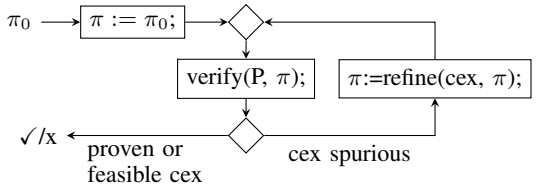


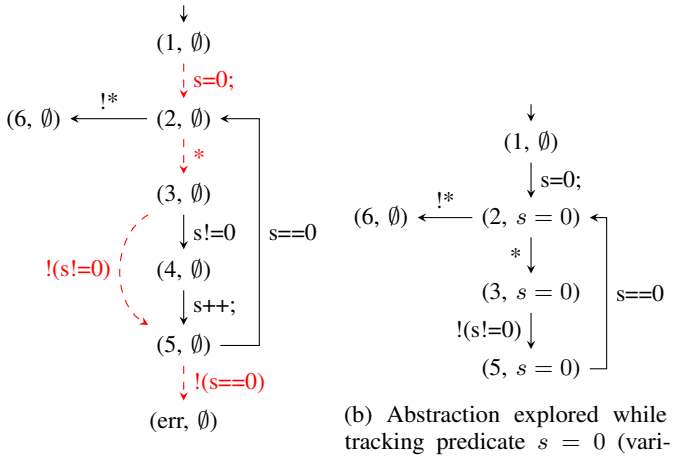
Fig. 1: Workflow of abstraction-based verification

Next, we demonstrate abstraction-based verification with two instances relevant for this paper, namely predicate analysis [12], [13], which abstracts the program with a set of predicates, and value analysis [14], which tracks a subset of the program variables. For our demonstration, we use the following program, for which we prove that it does not violate its assertion. The token * in the program represents a non-deterministic choice.

Listing 1: Example program

```
1 s=0;
2 while(*) {
3     if(s!=0)
4         s++;  // change to s+=2;
5     assert s==0;
6 }
```


This is the author's version of the work accepted for publication in 2026 IEEE 42nd International Conference on Software Maintenance and Evolution (ICSME).



(a) Abstraction explored with the empty precision (iteration 1)

(b) Abstraction explored while tracking predicate $s = 0$ (variable `s`) at locations 2, 3, and 5 (iteration 2)

Fig. 2: Abstractions explored in the different iterations of predicate analysis (value analysis) when applied to our example program (Listing 1)

Both, the predicate and the value analysis, start their verification with the coarsest abstraction precision, i.e., the empty precision, which does not consider any predicates and program variables, respectively. Hence, their initial abstract model (see Fig. 2a) only tracks the program's control-flow[1] but no information on the data state. While analyzing that model, both analyses detect a counterexample, a path from the program start (location 1) to a violation of the assertion (location err) that we mark with dashed, red lines in Fig 2a. This path is not feasible in the program due to a contradiction of `s` having the value zero at line 5 and the condition `!(s==0)`, which would need to be fulfilled to violate the assertion in line 5. Thus, both analyses refine their abstraction precision such that in the next iteration the predicate analysis tracks the predicate `s=0` at locations 2, 3, and 5, while the value analysis tracks the value of variable `s` at those locations. Figure 2b shows the resulting abstract model explored by the two analyses in their second iteration. We observe that the model no longer violates the assertion. Therefore, the analyses stop with the result that the program fulfills the given correctness criterion with no assertions violated.

Verifying a program once is not enough. As codebases evolve, changes may introduce regressions and, thus, may invalidate a previous verification result. Therefore, we need to (re)verify every new program version. However, performing a complete reverification for each change is too costly to fit within the constraints posed by iterative software development practices [15], such as continuous integration and delivery (CI/CD). While there exist some solutions [16], [17], [18], [19], [20], [21], [22] that address this performance problem for abstraction-based verifiers, only few abstraction-based verifiers like BLAST [16], CPACHECKER [17], [19], [20], [21], [22], and ULTIMATEAUTOMIZER [18], [21] implement any of those approaches. Precision reuse [17] is one of those solutions. Its evaluation in 2013 shows significant performance improvements (CPU time, number of refinements) for reverification with CPACHECKER and has been advertised as easily realizable and widely applicable to abstraction-based verifiers. Therefore, we think it may be an interesting approach that is worth integrating into other abstraction-based verifiers.

Precision reuse [17] builds on the observation that one often requires similar abstract precisions to verify the two subsequent versions P_1 and P_2 of a program. To speed up reverification of program (version) P_2, *precision reuse* therefore starts the iterative computation shown in Fig. 1 with the latest abstraction precision computed during the verification of program (version) P_1 instead of a coarse initial abstraction precision. For example, assume that we change line 4 of our example program (Listing 1) from `s++;` to `s+=2;` (as indicated by the comment). Instead of starting with the empty precision, which does not consider any predicates and program variables, respectively, precision reuse lets the predicate analysis (value analysis) start with the last abstraction precision used by our verification of our example program, namely the precision considered in iteration 2 that tells us to track predicate $s = 0$ (variable `s`) at locations 2, 3, and 5. For our example, this allows the analyses to explore the abstraction shown in Fig. 2b already in their first iteration and, thus, prove that the changed program does not violate its assertion. Compared to a reverification starting with the empty precision, precision reuse saves the first iteration in the verification of our modified example.

Before we encourage the software verification community to support precision reuse, which may even foster tool cooperation in context of reverification, we aim to investigate today's value of precision reuse in a reproduction and replication study. We perform a partial reproduction study, in which we did not involve the original authors, to examine whether the results from 2013 remain valid for CPACHECKER in 2025, even though the tool underwent significant (algorithmic) advances (see e.g., [23], [24], [25]). Moreover, we investigate in a partial, external replication whether the results from 2013 generalize, i.e., whether the results from 2013 carry over when using a different class of programs, in particular programs from a recent robustness study [26], or another abstraction-based verifier, more concretely the verifier THETA [7]. Our study investigates all claims examined by the original paper, thereby confirming the major claims. Also, we investigate and mostly confirm four claims stated in but not explicitly examined by the original paper. In general, our study observes precision reuse to be less beneficial than in 2013.

II. REVISITING REALIZABILITY CLAIMS

First, let us examine the claims regarding easy realizability and broad applicability of precision reuse, which we refer to as the realizability claims. The original paper on precision reuse [17] states the following three realizability claims.

RC 1 Integrating precision reuse into abstraction-based verifiers with automatic computation of the abstraction level is easy.

[1] We use the program's line numbers to refer to program locations.

RC 2 The precision format is easy to write and parse.

RC 3 Abstraction precisions are tool-independent and the proposed precision format can be supported by different verifiers.

Note that the original paper on precision reuse states the above claims but did not explicitly investigate them. To examine claims RC 1–3, we briefly recap the exchange format, the existing implementation of precision reuse in CPAchecker, and our implementation of precision reuse in Theta.

Proposed Exchange Format for Abstraction Precisions. The proposed exchange format is text-based and aims to be tool-independent. Independent of the precision type (predicates or tracked variables), the format groups the precision information (e.g., predicates) by scope (either global, function or location scope). The scope determines at which program locations the precision information is relevant, namely, in global scope at every program location, for function scopes at every location in the specified function, and for location scopes at the specified location. The format describes the scope by token ⋆ (global), the function name, or a tool-dependent location identifier. Under each scope, the format lists the precision information. To describe the variables the value analysis tracks in a particular scope, we list each of their scoped names in a separate line under the respective scope identifier. For example, CPAchecker tracks the parameter `__VERIFIER_assert::cond` in function scope `__VERIFIER_assert` (see Fig. 3a) while Theta tracks the same variable in global scope ⋆ (see Fig. 3b). In contrast, the format describes predicates (over program variables) that the predicate analysis considers in the standardized SMT-LIBv2 [27] format. At the top of a predicate precision file, a header introduces term declarations, which e.g., define the variables. Then, the file lists for each scope the relevant predicates, writing in each line one predicate in form of an SMT assertion. For instance, CPAchecker tracks predicate `main::s==0` at location N17 of function `main` (see Fig. 3d) and Theta tracks predicate `main::s%4294967296≤0` in global scope, i.e., at every location (see Fig. 3c).

Implementation of Precision Reuse in CPAchecker. CPAchecker exports the precision of the value analysis and the predicate analysis when outputting their statistics. The export is implemented with at most a few hundred lines of hand-written code. In case of the predicate analysis, the export relies on the used SMT solver to convert the predicates into Strings in SMT-LIB format. Parsing the precision files into CPAchecker's internal precision format is similarly simple, requiring also only a few hundred lines of hand-written code, plus the capabilities of the SMT solver to parse SMT-LIB definitions to extract the predicates.

Implementation of Precision Reuse in Theta. We implement precision reuse in Theta aiming at closely matching the proposed exchange format. Since Theta maintains precision information globally, it exports all precision information into the global scope ⋆ and ignores scope information during parsing. To read and write predicates, Theta relies on its existing functionality to parse and write SMT-LIBv2 code. Relying on the existing functionality, the implementation for precision reuse in Theta consists of a couple hundred lines of Kotlin code, which we developed in less than a workweek. However, making the implementation tool-independent takes some additional effort. For example, Theta uses `$$` for scoping of variables instead of `::` and only CPAchecker's predicate precisions quote variables with `|`. Also, CPAchecker's predicate precision uses `(set-info)` commands in the header, which are not expected nor necessary in the format. More difficult or impossible to resolve are tool-dependent internal variables and non-standard location identifiers in the exported precisions.

```
__VERIFIER_assert:
__VERIFIER_assert::cond

main:
main::s
```

(a) CPAchecker value analysis

```
*:
main::s
__VERIFIER_assert::cond
```

(b) Theta value analysis

```
(declare-fun |__VERIFIER_assert::cond| () Int)
(declare-fun |main::s| () Int)

*:
(assert (≤ (mod |main::s| 4294967296) 0))
(assert (= 0 |__VERIFIER_assert::cond|))
```

(c) Theta predicate analysis

```
(declare-fun |__VERIFIER_assert::cond| () (_ BitVec 32))
(declare-fun |main::s| () (_ BitVec 32))

__VERIFIER_assert:
(assert (let ((.def_63 (= |__VERIFIER_assert::cond|
                          (_ bv0 32)))).def_63))

reach_error N1:
(assert false)

main N17:
(assert (let ((.def_73 (= |main::s| (_ bv0 32)))).def_73))
```

(d) CPAchecker predicate analysis

Fig. 3: Precision file format used by the tools for loop-invariant/const.c

Conclusion on Realizability Claims. Based on our observations regarding CPAchecker's implementation and our experience with integrating precision reuse in Theta, we confirm that integrating precision reuse is easy (RC 1) and that the precision reuse format is easy to write and parse (RC 2). In contrast, we only partially confirm claim RC 3. The two implementations of precision reuse in CPAchecker and Theta show that different verifiers may support the format. However, although we tried to export precisions in Theta as close as possible to those exported by CPAchecker, we observed that internal differences of the tools transfer into exported precisions. This makes the exported precisions tool-dependent, thus, impeding exchange between tools. Hence, future work is needed to standardize the precision format.

III. Revisiting Performance Claims

In this section, we experimentally investigate the claims regarding the impact of precision reuse on performance and

resource consumption, i.e., the performance claims. Reading the paper on precision reuse [17], we extract the following five performance claims.

PC 1 Reusing abstraction precisions may increase effectiveness, i.e., a verifier may solve more tasks, especially additional tasks, when reusing precisions.

PC 2 Reusing abstraction precisions significantly reduces the number of required refinements of the abstraction level.

PC 3 Reusing abstraction precisions significantly reduces the verification time.

PC 4 Reusing abstraction precisions is efficient, i.e., the runtime overhead caused by parsing precisions is low.

PC 5 Abstraction precisions are small and require only a few kilobytes to store.

A. Our Setup in Comparison to the Original Setup

Verification Tasks. We use two sets of verification tasks, one for reproduction and one for replication. Both sets contain tasks that consists of a C program plus a safety property stating that a call to function `unreach-call` is never reachable.

For reproduction, we consider the *Linux tasks* from the original paper [17]. These tasks result from weaving monitor specifications into different versions of Linux device drivers. Thereby, each task belongs to one driver and one specification. All tasks belonging to the same driver and specification form a sequence of verification tasks, whose order is determined by the commit history of the driver. Tasks – except for the first task of a sequence – are modified tasks with the predecessor in their sequence being their previous version. Instead of only using those Linux tasks that fulfill their property, we take all 4 296 Linux tasks into account including the 103 tasks with property violations excluded in the original paper. Also, we use their adapted version from the `SVBenchmark`, which now uses entry function `main` and encodes the property violation as a call to function `unreach-call`.

For our replication study, we want to use tasks that are not considered by the original paper. Thus, we select the robustness tasks created for robustness testing of verification tools [26], a recent set of evolving verification tasks with specifications and expected verdicts. The *robustness tasks* consist of 778 original tasks (550 fulfill their property and 228 violate it) plus one to five versions per task, which each results from applying 100 semantics-preserving transformations. Such changes are to be expected in evolving codebases as a result of code maintenance activities (e.g., refactoring). In total, we consider 3 638 modified versions (2 546 without and 1 092 with property violations). The previous version of a modified version of a task is the respective original task.

Tool Configurations. Like the original paper on precision reuse [17], we consider CPAchecker's predicate analysis [13] (CPApred) and (explicit-)value analysis [14] (CPAval). Instead of using CPAchecker version 8112 from 2013 with SMT solver MathSAT 5.2.5, we use CPAchecker version 4.1 from August 2025 with MathSAT 5.6.10. In 2013 as well as in 2025, CPAchecker's predicate analysis employs predicate abstraction based on CEGAR [11], interpolation [28], lazy abstraction [29], and adjustable block encoding [13] configured to abstract at loop heads only. However in 2025, CPAchecker's predicate analysis uses a bit-precise encoding instead of a linear encoding and default options may be different, e.g., the transfer relation of the predicate analysis no longer performs satisfiability checks when reaching a property violation. Moreover, we enable the support for memory functions of the standard library and that memory allocation may be assumed to not fail, which did not exist in 2013 but are relevant for our verification tasks. In contrast, CPAchecker's value analysis changes conceptually. In 2025, it still uses CEGAR and interpolation [14] as in 2013, but additionally uses path prefix slicing [24] and refinement selection [25]. Finally, neither analysis enables the unsound option to skip recursions and both reuse precisions function-wide, i.e., precision information provided in global (function) scope remains relevant at all locations of the program (in the specified function), while location-scoped precision information is not only tracked at the particular location but at all locations that belong to the same function as the specified one.

Next to the two CPAchecker analyses, we consider two similar analyses in the verifier Theta, which use CEGAR with predicate (Thetapred) and explicit abstraction (Thetaval) [30], respectively. In contrast to CPAchecker, Theta uses global-scoped precisions, i.e., it always tracks all precision information (predicates or tracked variables) at every program location. In our configuration, Theta discards the previously explored state space and starts from scratch after each refinement step in its CEGAR procedure. We use Theta version 6.19.1 [31] extended with precision reuse in commit `92285273` together with SMT solver Z3 4.5.0.

We run all four analyses in two modes, no-reuse (nr) and with reuse (wr). In mode no-reuse, the analyses do not reuse any precision information, i.e., they start with the empty precision, the coarsest precision that typically causes all information to be abstracted. In mode reuse, an analysis only starts with the empty precision for the original task (i.e., first task in the sequence of versions). For the remaining versions, i.e., the modified tasks, the analysis reuses the precision it generated when analyzing the previous version of the task in mode reuse. For the robustness tasks, the previous version of any modified task is the original task, while for the Linux tasks, the corresponding sequence of verification tasks determines the previous task. More concretely, for a sequence $(s_1, \ldots, s_n)$ of Linux tasks, the previous version of a modified task s_i $(2 \leq i \leq n)$ is s_{i-1}.

Environment. We perform our experiments with more recent hardware and software but aim to use the same resource limits. Our machines use an Intel Xeon E3-1230 v5 CPU instead of an Intel Core i7-2600 CPU but use the same amount of RAM (32 GB) as the machines in the original paper. Also, our machines run an Ubuntu 24.04 with Linux 6.8.0 instead of an Ubuntu 12.04 with Linux 3.2. Like the original paper, we limit each verification run, i.e., the execution of a tool (configuration) on a verification task, to 15 min of CPU time, 15 GB of RAM, and 4 CPU cores. We enforce the limits with BenchExec (version 3.30) [32].

TABLE I: Number of (in)correct alarms and proofs the respective configuration reports in mode no-reuse (nr) and with reuse (wr) for the respective task set

	Linux tasks				robustness tasks							
	CPApred		CPAval		CPApred		CPAval		ThetaPred		ThetaVal	
	nr	wr	nr	wr	nr	wr	nr	wr	nr	wr	nr	wr
correct proofs	3203	3203	3902	3902	376	410	38	39	251	245	51	55
correct alarms	71	69	56	56	167	172	254	289	93	113	95	96
incorrect proofs	3	3	0	0	0	0	0	0	6	6	8	8
incorrect alarms	0	0	0	0	0	0	0	0	0	0	0	0

B. Revisiting PC 1 – Increase in Effectiveness

To validate the claim that precision reuse may increase the effectiveness, we study the number of modified tasks solved by each configuration with and without precision reuse. Table I shows for each of CPAchecker's configurations the number of proofs and alarms the configuration reported (in)correctly with respect to the modified Linux tasks as well as the number of proofs and alarms all four configurations (in)correctly report for the modified robustness tasks.[2]

First, let us focus on reproduction, i.e., we consider the first four columns of Tab. I, which shows the effectiveness of CPAchecker on Linux tasks. We observe that the numbers are almost identical for configurations with and without reuse. A detailed analysis reveals that CPAchecker's value analysis with reuse solves the same tasks as that configuration without reuse while CPAchecker's predicate analysis without reuse solves two additional task, for which its configuration with reuse times out. These two tasks violate their property and are excluded from the experiments in the original paper. Compared to the results reported in the original paper on precision reuse [17], we notice that the number of solved tasks decreases in general and the configurations with reuse no longer solve additional tasks. The latter can be attributed to improved analyses which are now also able to solve tasks without precision reuse that required precision reuse in 2013. The decrease of the total number of solved tasks is because we do not skip recursion, which is unsound, and because CPAchecker's predicate analysis now uses a bit-precise encoding, which does not support all features of the Linux tasks and, thus, may cause the predicate analysis to fail with an error.

Next, let us continue with the replication aspect and the robustness tasks. We start with the tasks solved by the CPAchecker configurations (columns 5–8). Both configurations detect more proofs and alarms with precision reuse. Also, the configurations with precision reuse mostly solve all tasks their respective configurations without precision reuse solve. There exists only one task that CPAval solves without precision reuse and five tasks that CPApred solves without precision reuse, while they timeout with precision reuse.

Looking at the tasks solved by Theta's configurations (last four columns), we notice that ThetaVal solves more proofs and alarms with precision reuse. A detailed inspection of the results shows that ThetaVal with precision reuse solves all tasks that ThetaVal without precision reuse solves. In contrast, ThetaPred with reuse solves more alarms but less proofs. Thereby, ThetaPred with precision reuse detects all alarms that ThetaPred without precision reuse detects. However, ThetaPred detects some proofs only when reusing precision and some only when not reusing precisions. Typically, the different results are caused by an error in the verification process (e.g., a solver error). In the majority of the cases, this is due to an internal solver error of the Z3 version that was used during verification. A newer Z3 version enables Theta to solve these tasks.

[2]An incorrect proof misses an alarm, while an incorrect alarm issues a false warning.

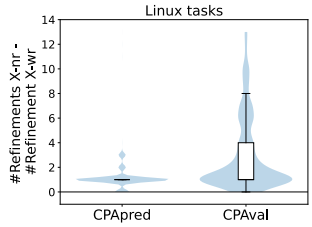


Fig. 4: For Linux tasks and CPAchecker analyses, distribution of the difference in refinements used per configuration when not and when reusing precisions

Result. While changes in CPAchecker's analyses may not allow us to reproduce the results from 2013, our insights on the robustness tasks let us confirm the claim, precision reuse may increase effectiveness. Nonetheless, precision reuse may sometimes negatively affect the effectiveness.

C. Revisiting PC 2 – Less Refinements

In this section, we examine the claim that precision reuse significantly reduces the number of applied refinements, i.e., the number of times the workflow in Fig. 1 refines the current abstraction. Therefore, we compare for each configuration the number of refinements the configuration requires with and without precision reuse. We restrict the comparison to all modified tasks that the configuration solves with and without precision reuse.

First, let us focus on reproduction. Figure 4 shows two combined violin and box plots, which show the distributions of the difference in the applied refinements for CPApred (left) and CPAval (right) without and with precision reuse considering 3 272 Linux tasks and 3 958 Linux tasks, respectively. We observe that the difference is at least zero, i.e., the configuration with precision reuse never performs more refinements. Also, the distribution is larger for CPAval, i.e., it may save more

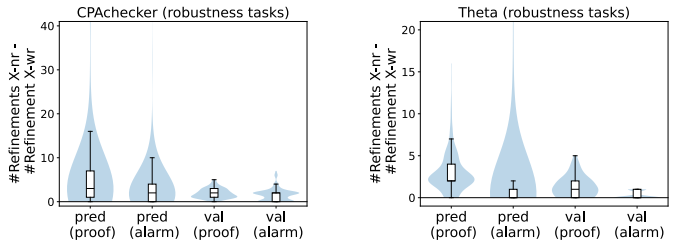


Fig. 5: For robustness tasks and different tools, distribution of the difference in refinements used per configuration when not and when reusing precisions

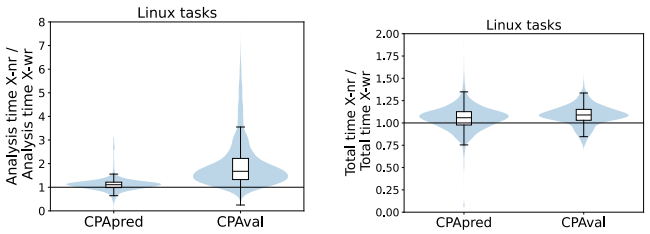


Fig. 6: Distribution of the speedup of the analysis time (left) and the total CPU time (right) that CPACHECKER's analyses achieve when reusing precisions

refinements than CPAPRED. A detailed analysis reveals that most of the times, reusing precisions reduces the number of performed refinements. Applying the Wilcoxon signed-rank test reveals that the reduction in the number of refinements is statistically significant with a p-value of less than 4.94×10^{-324}. In 88% of the tasks (2 895 of 3 272) and 93% of the tasks (3 679 of 3 958), CPAPRED and CPAVAL perform less refinements when reusing precisions. In the median, we save one refinement when reusing precisions. More importantly, we do not refine (i.e., perform zero refinements) for more than 90% of the tasks when reusing precisions. Like in our introductory example, for these tasks the precision used to verify the previous version is sufficient to verify the current program version. To compare our results with the results reported in 2013 [17], we also compute the accumulated number of refinements per pair of device driver and specification. We observe that for CPAVAL we achieve the same number of refinements, both in mode no-reuse and mode with reuse, as in 2013 for about 70% of the pairs while for CPAPRED the results are rarely identical. One reason for the latter is that in 2025 CPAPRED often uses less refinements in mode no-reuse than in 2013, i.e., CPAPRED's refinement procedure improved.

Next, we study whether we may replicate the difference in refinements with the robustness tasks. To this end, the left and right graphics of Fig. 5 each show four combined violin and box plots. The four plots in the left show the distributions of the difference in the performed refinements for CPAPRED (left two plots) and CPAVAL (right two plots) without and with precision reuse. Thereby, each configuration has one plot that only considers modified robustness tasks without property violations (proofs) and one that only considers modified robustness tasks with property violations (alarms). CPAPRED considers 371 proofs and 167 alarms while CPAVAL considers 38 proofs and 253 alarms. Similarly, the four box plots on the right of Fig. 5 show the distributions of the difference in the applied refinements for THETA's predicate and value analysis. These plots consider 238 proofs and 93 alarms for THETAPRED and 51 proofs and 95 alarms for THETAVAL. Again, we observe that the differences are always above zero, i.e., the configurations with precision reuse never perform more refinements. Also, the difference in the number of refinements tends to be larger for tasks without property violations (i.e., proofs). The Wilcoxon signed-rank test confirms that the reduction in the number of refinements is statistically significant ($p < 1.49 \times 10^{-12}$). A more detailed investigation reveals that THETAPRED and THETAVAL typically require the same number of refinements (median and mean difference of zero) in mode no reuse and with reuse when detecting property violations. In contrast, THETAVAL saves one refinement in the median while THETAPRED saves two refinements in the median when detecting proofs and typically do not require any refinements with precision reuse when detecting proofs. Interestingly, CPAVAL with precision reuse saves around two refinements in the median independently of whether it shows the existence or absence of a property violation. Similarly, CPAPRED with precision reuse saves around two refinements in the median when detecting property violations and around three refinements in the median when proving the absence of property violations. Also, CPAVAL and CPAPRED typically do not require any refinements with precision reuse to show that a task adheres to its property.

Result. We confirm that if we prove the absence of property violations, the number of refinements decreases when reusing precisions. Often, we do not even need any refinements but some refinements may be needed if changes affect variable names or code is moved to new functions. Also, advances in the analyses, which reduces the number of performed refinements, make the difference with respect to the number of refinements between not reusing and reusing precisions smaller than in 2013. In contrast, when detecting property violations, which was not studied in 2013, it depends on the analysis and verification task whether one may benefit from precision reuse. All in all, precision reuse may have positive effects but never has negative effect on the number of refinements.

D. Revisiting PC 3 – Reduced Verification Time

Now, we examine the claim that reusing precisions significantly reduces the verification time. Similar to the original paper [17], we study the effect of precision reuse on the analysis time (i.e., the time the verification tool spends in the actual analysis of the verification tasks excluding tool start times and time for program parsing). Since THETA can only measure the analysis time in wall time, we decide to measure analysis time in wall time not CPU time as done in the original paper on precision reuse [17]. Also, we compare the total CPU time a

configuration requires to successfully verify a task with and without precision reuse.

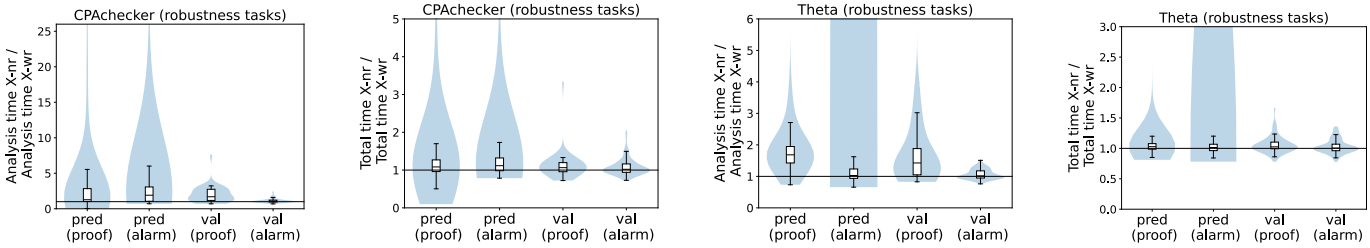
As before, we start with the reproduction aspect. Figure 6 shows two graphics that each contain one combined violin and box plot for CPAPRED (left) and one for CPAVAL (right). The left graphic shows the distribution of the speedup of the analysis time the respective analysis achieves when reusing precisions in relation to not reusing precisions. Similarly, the right graphic shows the distribution of the total time taken by the verification tool. Like the refinement comparison, we only consider modified Linux tasks that the respective configuration solves with and without reuse. When looking at the plots, we observe that most parts of the violins and boxes are above one, i.e., reusing precisions makes the analysis faster. Moreover, speedups for the total CPU time are smaller than for the analysis time, which is not surprising because we add a constant overhead to the analysis time. We may achieve larger speedups for CPAVAL when considering the analysis time only, but the achievable speedups for the total time are similar for both analyses. A detailed analysis reveals that precision reuse speeds up CPAPRED in about 70% of the tasks (2 378 and 2 262 of 3 272), thereby accelerating the analysis by 10% and the verification by 6% in the median. For CPAVAL, precision reuse speeds up about 80%–95% of the tasks (3 766 and 3 243 of 3 958) with a median speedup of 1.7 on the analysis time and 1.09 on the total time. We observe that many slow downs occur if the verification takes less than 20 seconds[3]), and we further notice that when no speedup is achieved, the difference between the time without and with precision reuse is insubstantial (less than 1.5 seconds in the median). Also, the variance of the analysis and total time is lower in mode with reuse.

To compare our results with the results reported in 2013 [17], we must aggregate the achieved speedups per pair of device driver and specification. To this end, we compute for each pair the arithmetic mean of the speedups of all modified tasks that are solved with and without precision reuse and that belong to the pair. For the analysis time, we observe that the maximum average speedup and the average of the average speedups over all pairs decreases for CPAPRED from 63 to 2.2 and 4.3 to 1.08, respectively, while there exist more pairs with a slowdown, namely 58 instead of 6. Similarly, the maximum average speedup and the average of the average speedups decreases for CPAVAL, namely from 210 to 5.1 and from 3.8 to 1.76. However, there exist less pairs with a slowdown, namely 6 instead of 7. This observation carries over when looking at the sum of the total times of all successfully verified tasks. In 2013 [17], CPAPRED requires 99 000s and 38 000s of CPU time to solve 4 048 and 4 078 tasks without and with precision reuse (i.e., a speedup of 2.6). In contrast, CPAPRED now requires about 99 000s and 97 000s of CPU time to solve only 3 274 and 3 272 tasks without and with precision (i.e., a speedup of 1.02). Similarly, CPAVAL requires 28 000s and 20 000s of CPU time to solve 4 193 tasks without and with reuse in 2013 (i.e., a speedup of 1.4). In contrast, CPAVAL now requires about 42 000s and 40 000 s of CPU time to only solve 3 958 tasks without and with precision reuse (i.e., a speedup of 1.05). Since the analyses rarely perform any refinement when reusing precisions, the exploration of the abstract state space must have become more costly, e.g., due to a more precise encoding. Also, the time lost due to iterative refinements becomes smaller in case of a smaller number of refinement (i.e., iterations in Fig. 1). We also observed that less of the total time is spent on the analysis. Thus, the fraction of time precision reuse can save becomes smaller, resulting in lower speedups. Applying the Wilcoxon signed-rank test to the tasks that were solved both in 2013 and 2025 without and with precision, we find that the time reduction is still statistically significant with p-values of at most 9.43×10^{-308}. However, the effect size r is reduced from a strong 0.77 in 2013 to a moderate 0.44, reflecting the reduced benefits.

Next, we continue with the replication aspect and investigate whether the verification time significantly reduces when applying precision reuse to CPACHECKER's and THETA's analyses on the robustness tasks. Figure 7 shows four graphics, two for CPACHECKER (left) and two for THETA (right). Each graphic consists of four combined violin and box plots that show the distribution of the speedups the tool's predicate and value analyses achieve with precision reuse when detecting proofs and alarms, respectively. Graphics 7a and 7c show the speedups of the analysis time (without program parsing, etc.), while graphics 7b and 7d show the speedups of the total time taken by the verification tool. All plots only consider tasks that the respective configurations solves in mode no-reuse and in mode with reuse. Looking at the plots, we once more observe that most parts of the violins and boxes are above one, i.e., reusing precisions makes the analysis faster. Also, speedups for the total CPU time are smaller than for the analysis time. However, it seems to depend on the tool and its analysis how large speedups may become. Depending on the analysis, precision reuse speeds up the analysis and total time for 77-93% and 63-66% of the tasks when detecting proofs. Applying the Wilcoxon signed-rank test, we confirm the time reduction is statistically significant ($p < 2.61 \times 10^{-9}$). Since for many of these tasks we do not need any refinements when reusing precisions, we observe the full potential of precision reuse. When detecting alarms precision reuse speeds up the analysis and total time less often, namely for 54-80% and 54-56% of the tasks. Regardless, the Wilcoxon signed-rank test still confirms that the time reduction is statistically significant ($p < 3.74 \times 10^{-2}$). In general, slowdowns typically occur if the verification takes less than 20 seconds and we further notice that when no speedup is achieved, the difference between the time without and with precision reuse is less than a second in the median. While we may achieve speedup factors of almost one thousand, the median speedup of an analysis is often in the range of 1.01 to 1.1 but not larger than two. Except for CPAVAL, the variance of the analysis and total time is lower in mode with reuse, too.

Result. We partially confirm the claim. For a significant portion

[3]A verification time above 20s was one of the criteria when generating the Linux tasks [17].



(a) CPAchecker analysis time (b) CPAchecker total CPU time (c) Theta analysis time (d) Theta total CPU time

Fig. 7: Distribution of the speedup of the analysis time (a, c) and the total CPU time (b, d) that CPAchecker's analyses (left) and Theta's analyses (right) achieve when reusing precisions to show the absence (proofs) or existence of property violations (alarms) for robustness tasks

of tasks, precision reuse reduces the verification time. For many tasks where the analyses prove the absence of property violations, we require zero refinements and, thus, observe the maximal potential of precision reuse for that task. While we observe high speedups for some tasks, often the speedup is not much larger than 10%, which we do not consider significant, as it is orders of magnitude smaller than the original average speedup of 380–430%. In general, it depends on the analysis and the task how large the speedup is or whether there is any. Since we observe most slowdowns on tasks with short verification times (e.g., below 20s[4]), precision reuse may work better on longer running tasks. Also, lower speedups for the total time indicate that it may be beneficial if the analysis time dominates the total verification time. In addition, a higher number of refinements may be beneficial. Nevertheless, many slowdowns are negligible with a difference in verification time of less than one second. Once again, precision reuse is not really harmful but may have positive effects.

E. Revisiting PC 4 – Efficient Parsing of Precisions

Next, we look into a claim stated but not investigated in the original paper, namely whether abstraction precisions are efficient to reuse, i.e., the costs for parsing them are low. Since CPAchecker does not provide time measurements for precision import and export, we only study the wall time Theta spends on importing (i.e., parsing) and exporting (i.e., writing) precisions. The left of Fig. 8 shows two combined violin and box plots that show the distributions of the wall time Theta's predicate and value analysis spend on importing (i.e., parsing, reading) precision files when verifying modified robustness tasks in mode with reuse. Similarly, the right of Fig. 8 shows two combined violin and box plots that show the distributions of the wall time Theta's predicate and value analysis spend on exporting (i.e., writing) precision files when verifying the original tasks from the robustness tasks. Note that all plots only include wall times for a subset of the tasks because Theta typically does not output the respective wall time when failing. The two plots on the left consider 361 and 160 tasks and the two plots on the right consider 273 and 100 tasks. Looking at the plots, we observe that importing and exporting precisions only takes a few milliseconds. The time it takes to export the value precision is often not even measurable by Theta. In general, the import and export of predicate precisions are more costly than the import and export of value precisions. This is because value precisions are smaller in terms of file size (see next section) and their export and import does not require a transformation to SMT-LIBv2. Compared to the analysis time and the total time, writing and parsing precisions is fast.

Result. We confirm the claim that the additional computation effort required to reuse precisions is low and negligible in context of the verification time.

F. Revisiting PC 5 – Small Precisions

At last, we examine the claim that abstraction precisions are small and require only a few kilobytes to store. To this end, we look at the exported precision files, which store the final precision used by a tool configuration. Table II shows for all four analyses and all task sets they are run on, the average and maximum sizes of the generated precision files in bytes.

First, let us focus on reproduction (first two columns of Tab. II). We observe that CPAval produces smaller precisions files than CPApred and that CPAval's precision files are small with around 100 bytes on average and less than one kilobyte at worst. Also, CPApred's precision files are small on average, i.e., below one kilobyte on average. However, the largest precision file CPApred generates is much larger with about 43 kilobytes. Still, the largest precision file remains significantly smaller than the file size of the verification tasks.

When comparing our results with the results from 2013 [17], we forego a detailed comparison of the results and only compare average and maximum file sizes per analysis configuration. We observe that precision files for CPAval remain smaller than for CPApred. Also, we notice that for CPApred the average file size for precisions decreases from 1 kilobyte to 0.85 kilobytes. However, the maximal size of a precision file increases from 4 to 43 kilobytes. For CPAval, the average (maximum) file size for precisions decreases from 110 bytes (1 kilobyte) to 101 bytes (0.73 kilobytes). We believe that one major reason for the reduction in file size is the decrease in refinements. Considering that each refinement will add at least one new

[4]A verification time above 20s was one of the criteria when generating the Linux tasks [17].

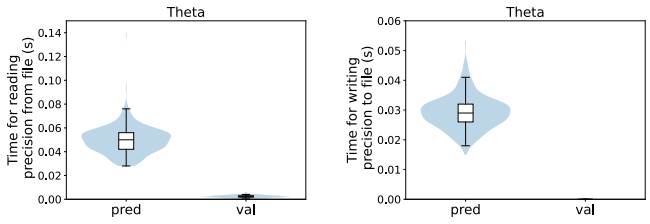


Fig. 8: Distribution of the wall time THETA's configurations spend on reading (left) and writing (right) precisions

predicate or variable to track, we think that a lower number of refinements may be an indicator that a coarser precision suffices to solve a verification task. Also, storing coarser precisions (i.e., less tracked predicates or variables) should lead to smaller file sizes. One reason for the increase in the maximum file size in context of CPAPRED might be that predicate analysis uses a bit-precise encoding, in which predicates might be more costly to encode in SMT-LIB.

Finally, we study whether the claim regarding the size of the precision files carries over to the robustness tasks (last four columns of Tab. II). Again, we observe that precision files are smaller for value analyses (CPAVAL, THETAVAL). Storing a precision of the value analyses requires at maximum one kilobyte. Also, the average file size of precisions of the predicate analyses is in the range of a few kilobytes (16 kilobytes for CPAPRED and 2 kilobytes for THETAPRED). However, for CPAPRED and THETAPRED the maximum size of the precision files is orders of magnitude larger, namely 100 kilobytes for THETAPRED and even 4 megabytes for CPAPRED, which is much larger than many of the robustness tasks.

Result. We confirm that precisions for value analyses are small and in the range of a few kilobytes. However, if we need to store predicate precisions we require more space. While a predicate precision can often be stored within a few kilobytes, we also observe that predicate precisions may become significantly larger and in particular considerably larger than the program itself. Thus, predicate precisions are often but not always small. Hence, we only partially confirm the claim.

G. Conclusions on Performance Claims

We completely confirm two of the performance claims, namely precision reuse may increase effectiveness (PC 1) and the costs for precision parsing is low (PC 4). In addition, we partially confirm the remaining three performance claims. Their major claims still tend to apply but sometimes they only apply to many of the verification tasks and the reduction effect with respect to the number of refinements and verification time is less significant. Importantly, we also observe positive effects when detecting property violations, a setting not studied in the original paper [17]. Furthermore, negative effects often do not exist or are minor. For example, parsing large precision files does not seem to have a large impact on the verification time. In general, the verification task itself as well as the

TABLE II: Per evaluated combination of configuration and task set, shows the average and maximum file size in bytes of the generated precision files

file size	Linux tasks		robustness tasks			
	CPAPRED	CPAVAL	CPAPRED	CPAVAL	THETAPRED	THETAVAL
avg.	867	101	16102	63	1741	37
max.	43685	751	3851104	276	107430	909

employed analysis seem to influence whether or not and how beneficial precision reuse is. When proving the absence of property violation, we often use the full potential of precision reuse. Still, this is not enough to be beneficial. The time saved by skipping iterations of the verification workflow (see Fig. 1) must be significant. As a proxy, one may use the number of refinements required during verification, i.e., a higher number of refinements for verifying the original program may be more beneficial. Also, the proportion of the analysis time on the total time is important, i.e., the time to set up the analysis including the time to parse a program has a significant effect on the speedup and should not be ignored. In particular, if the parsing time of the program dominates the total verification time, which can be the case for tasks that can be verified quickly, reducing the analysis time has little impact on the overall reduction of the verification time and one may need to combine precision reuse with incremental program parsing to get a visible speedup. Additionally, we observe most slowdowns on tasks with short verification times (e.g., below 20s[5]). Precision reuse is likely more beneficial for tasks with longer verification times. Note that the number of applied refinements, the proportion of analysis time on the total verification time and the time required to verify a task are all data that can be measured when verifying the original (unmodified) task and help to estimate whether precision reuse might be beneficial for one's application. Due to rare negative effects, one can also always apply precision reuse if it is available anyways, but then may not always profit from it.

IV. THREATS TO VALIDITY

Internal Threats. While we included objective criteria like lines of code (LOC) in our investigation of the easy realizability (claims RC 1 and 2), the evaluation may still be biased by our subjective assessment as domain experts. Non-domain experts may come to a different conclusion. Moreover, LOC is not a very good metric for simplicity. Another threat is time measurements. Due to technical reasons, we cannot perform all time measurements with an external, reliable infrastructure like BENCHEXEC. In particular, the verifiers internally measure the analysis time as well as the time for importing and exporting precisions. Thus, these measurements may be imprecise. For claim PC 3, we mitigate this by additionally studying the total CPU time, which we measure externally and reliably with BENCHEXEC. Since times measured for import and export are

[5]Remember a verification time above 20s was one of the criteria when generating the Linux tasks [17].

orders of magnitude smaller than the total CPU time required for verification, we believe that imprecisions for measured times do not affect our results for claim PC 4.

External Threats. While we extended the verification tasks beyond Linux device drivers, we still only consider verification tasks for C programs. Thus, our results may not carry over to all real-world programs. Also, the results likely differ for other abstract domains than the predicate and value abstract domains. We in particular expect differences regarding efficiency, which we already observed for our two domains. In addition, we observe a difference in the performance behavior between the verifiers CPAchecker and Theta, even for the same domains. Hence, we expect that the results are different for other verification tools or future, enhanced versions of our verification tools CPAchecker and Theta. Despite the observed differences for CPAchecker and Theta, we individually confirm the major claims on precision reuse. Also, we are able to show that the major claims on precision reuse carry over to a recent CPAchecker version. Precision reuse may be less efficient or effective for another tool (version) but we expect the major claims to carry over. Moreover, we expect the results to be different when using other resource limits.

V. Conclusion

In 2013, Beyer, Löwe, Novikov, Stahlbauer, and Wendler suggested to export abstraction precisions (e.g., tracked predicates or variables) and then later import and reuse them to speed up the reverification of a modified program version [17] with abstraction-based verifiers. In this paper, we investigate in a reproduction and replication study whether the claims from the original paper [17] remain valid when using recent verifier technology and different verification tasks. Our study is one of the few reproduction and replication studies in software verification. We are only aware of two studies that reproduce software competition results [33], [34], one study that investigates an algorithm in context of concurrency verification [35], and two studies in statistical model checking [36], [37]. Our reproduction study uses a recent version of the verifier CPAchecker, the verifier used for evaluation in 2013, but relies on the original Linux task set. Our replication study extends the considered verifiers (we add Theta) and uses a different task set based on the tasks of a recent robustness study [26]. Our study confirms the major claims from 2013. However, we notice that the improvement regarding efficiency is less significant than in 2013 even when using the full potential of precision reuse (i.e., no refinements in mode reuse) and precisions are no longer always but only often small. More concretely, the verification task itself as well as the employed analysis influence whether or not and how beneficial precision reuse is. For example, the number of refinements required in mode no reuse but also the total time for verification, and the ratio of the total time that is spent on the analysis seem to play a role to achieve significant improvements and these metrics are often lowered when improving an analysis. Regarding the claims that have not yet been investigated, namely claims regarding the realizability and the parsing costs of precisions we attest that it is simple to integrate precision reuse into an existing verification workflow but notice that the exported precisions are at least partially tool-dependent. Despite some precisions being large, we also show that costs for writing and reading precisions are negligible. In general, we observe hardly any negative effects of precision reuse on a verifier's performance. Thus, we recommend to activate precision reuse when reverifying modified programs in evolving codebases to profit from its benefits whenever possible. Nevertheless, we need further studies that allows us to better understand and predict when precision reuse provides high benefits. Furthermore, we need to improve the tool-independency of the exported precisions if we want to exchange precisions between verifiers.

Data-Availability Statement. A reproduction package that includes all software and data that we used for our experiments is available on Zenodo [38].

Funding Statement. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 378803395 (ConVeY) and 496852682 (ReVeriX).

References

[1] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE TCAD*, vol. 27, no. 7, pp. 1165–1178, 2008. [Online]. Available: https://doi.org/10.1109/TCAD.2008.923410

[2] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys*, vol. 41, no. 4, 2009. [Online]. Available: https://doi.org/10.1145/1592434.1592438

[3] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proc. PLDI*. ACM, 2001, pp. 203–213. [Online]. Available: https://doi.org/10.1145/378795.378846

[4] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast," *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007. [Online]. Available: https://doi.org/10.1007/s10009-007-0044-z

[5] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Proc. CAV*, ser. LNCS 6806. Springer, 2011, pp. 184–190. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_16

[6] D. Baier, D. Beyer, P. Chien, M. Jakobs, M. Jankola, M. Kettl, N. Lee, T. Lemberger, M. L. Rosenfeld, H. Wachowitz, and P. Wendler, "Software verification with CPAchecker 3.0: Tutorial and user guide," in *Proc. FM*, ser. LNCS 14934. Springer, 2024, pp. 543–570. [Online]. Available: https://doi.org/10.1007/978-3-031-71177-0_30

[7] Á. Hajdu and Z. Micskei, "Efficient strategies for CEGAR-based model checking," *JAR*, vol. 64, no. 6, pp. 1051–1091, 2020. [Online]. Available: https://doi.org/10.1007/s10817-019-09535-x

[8] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate abstraction of ANSI-C programs using SAT," *Formal Methods Syst. Des.*, vol. 25, no. 2-3, pp. 105–127, 2004. [Online]. Available: https://doi.org/10.1023/B:FORM.0000040025.89719.f3

[9] ——, "SATABS: sat-based predicate abstraction for ANSI-C," in *Proc. TACAS*, ser. LNCS 3440. Springer, 2005, pp. 570–574. [Online]. Available: https://doi.org/10.1007/978-3-540-31980-1_40

[10] F. Schüssele, M. Bentele, D. Dietsch, M. Heizmann, X. Jiang, D. Klumpp, and A. Podelski, "Ultimate Automizer and the abstraction of bitwise operations - (competition contribution)," in *Proc. TACAS*, ser. LNCS 14572. Springer, 2024, pp. 418–423. [Online]. Available: https://doi.org/10.1007/978-3-031-57256-2_31

[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003. [Online]. Available: https://doi.org/10.1145/876638.876643

[12] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *Proc. CAV*, ser. LNCS 1254. Springer, 1997, pp. 72–83. [Online]. Available: https://doi.org/10.1007/3-540-63166-6_10

[13] D. Beyer, M. E. Keremoglu, and P. Wendler, "Predicate abstraction with adjustable-block encoding," in *Proc. FMCAD*. IEEE, 2010, pp. 189–197. [Online]. Available: https://ieeexplore.ieee.org/document/5770949/

[14] D. Beyer and S. Löwe, "Explicit-state software model checking based on CEGAR and interpolation," in *Proc. FASE*, ser. LNCS 7793. Springer, 2013, pp. 146–162. [Online]. Available: https://doi.org/10.1007/978-3-642-37057-1_11

[15] P. W. O'Hearn, "Continuous reasoning: Scaling the impact of formal methods," in *Proc. LICS*. ACM, 2018, p. 13–25. [Online]. Available: https://doi.org/10.1145/3209108.3209109

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido, "Extreme model checking," in *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, ser. LNCS 2772. Springer, 2003, pp. 332–358. [Online]. Available: https://doi.org/10.1007/978-3-540-39910-0_16

[17] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler, "Precision reuse for efficient regression verification," in *Proc. FSE*. ACM, 2013, pp. 389–399. [Online]. Available: https://doi.org/10.1145/2491411.2491429

[18] B. Rothenberg, D. Dietsch, and M. Heizmann, "Incremental verification using trace abstraction," in *Proc. SAS*, vol. LNCS 11002. Springer, 2018, pp. 364–382. [Online]. Available: https://doi.org/10.1007/978-3-319-99725-4_22

[19] F. He, Q. Yu, and L. Cai, "Efficient summary reuse for software regression verification," *TSE*, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2020.3021477

[20] Q. Yu, F. He, and B. Wang, "Incremental predicate analysis for regression verification," *TOPLAS*, vol. 4, no. OOPSLA, pp. 184:1–184:25, 2020. [Online]. Available: https://doi.org/10.1145/3428252

[21] D. Beyer, M. Jakobs, and T. Lemberger, "Difference verification with conditions," in *Proc. SEFM*, ser. LNCS 12310. Springer, 2020, pp. 133–154. [Online]. Available: https://doi.org/10.1007/978-3-030-58768-0_8

[22] M. Jakobs and T. Pollandt, "diffDP: Using data dependencies and properties in difference verification with conditions," in *Proc. iFM*, ser. LNCS 14300. Springer, 2023, pp. 40–61. [Online]. Available: https://doi.org/10.1007/978-3-031-47705-8_3

[23] D. Beyer, M. Dangl, and P. Wendler, "A unifying view on SMT-based software verification," *JAR*, vol. 60, no. 3, pp. 299–335, 2018. [Online]. Available: https://doi.org/10.1007/s10817-017-9432-6

[24] D. Beyer, S. Löwe, and P. Wendler, "Sliced path prefixes: An effective method to enable refinement selection," in *Proc. FORTE*, ser. LNCS 9039. Springer, 2015, pp. 228–243. [Online]. Available: https://doi.org/10.1007/978-3-319-19195-9_15

[25] ——, "Refinement selection," in *Proc. SPIN*, ser. LNCS 9232. Springer, 2015, pp. 20–38. [Online]. Available: https://doi.org/10.1007/978-3-319-23404-5_3

[26] F. Dyck, C. Richter, and H. Wehrheim, "Robustness testing of software verifiers," in *Proc. SEFM*, ser. LNCS 14323. Springer, 2023, pp. 66–84. [Online]. Available: https://doi.org/10.1007/978-3-031-47115-5_5

[27] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.5," University of Iowa, Tech. Rep., 2015. [Online]. Available: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-06-28.pdf

[28] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *Proc. POPL*. ACM, 2004, pp. 232–244. [Online]. Available: https://doi.org/10.1145/964001.964021

[29] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Proc. POPL*. ACM, 2002, pp. 58–70. [Online]. Available: https://doi.org/10.1145/503272.503279

[30] Á. Hajdu, T. Tóth, A. Vörös, and I. Majzik, "A configurable CEGAR framework with interpolation-based refinements," in *Proc. FORTE*, ser. LNCS 9688. Springer, 2016, pp. 158–174. [Online]. Available: https://doi.org/10.1007/978-3-319-39570-8_11

[31] Cs. Telbisz, L. Bajczi, D. Szekeres, and A. Vörös, "THETA: Various approaches for concurrent program verification (competition contribution)," in *Proc. TACAS*, ser. LNCS 15698. Springer, 2025, pp. 260–265. [Online]. Available: https://doi.org/10.1007/978-3-031-90660-2_22

[32] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: Requirements and solutions," *STTT*, vol. 21, no. 1, pp. 1–29, 2019. [Online]. Available: https://doi.org/10.1007/s10009-017-0469-y

[33] M. Gerhold and A. Hartmanns, "Reproduction report for SV-COMP 2023," *CoRR*, vol. abs/2303.06477, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.06477

[34] L. Bajczi, Zs. Ádám, and Z. Micskei, "SV-COMP'25 reproduction report (competition contribution)," in *Proc. TACAS*, ser. LNCS 15698. Springer, 2025, pp. 187–191. [Online]. Available: https://doi.org/10.1007/978-3-031-90660-2_10

[35] L. Bajczi, Cs. Telbisz, D. Szekeres, and A. Vörös, "On stability in a happens-before propagator for concurrent programs (reproducibility study)," in *Proc. TACAS*, ser. LNCS 15696. Springer, 2025, pp. 3–19. [Online]. Available: https://doi.org/10.1007/978-3-031-90643-5_1

[36] C. E. Budde and A. Hartmanns, "Replicating sc restart with prolonged retrials: An experimental report," in *Proc. TACAS*, ser. LNCS 12652. Springer, 2021, pp. 373–380. [Online]. Available: https://doi.org/10.1007/978-3-030-72013-1_21

[37] T. Quatmann, "What is the best algorithm for MDP model checking?" *STTT*, vol. 27, no. 4, pp. 431–437, 2025. [Online]. Available: https://doi.org/10.1007/s10009-025-00810-4

[38] M. Jakobs and M. Somorjai, "Reproduction package for the ICSME 2026 submission 'Revisiting precision reuse – a reproduction and replication study'," Zenodo, 2026. [Online]. Available: https://doi.org/10.5281/zenodo.20230042