Reuse of Verification Results
Conditional Model Checking, Precision Reuse, and Verification Witnesses

Dirk Beyer and Philipp Wendler

University of Passau, Germany

Abstract. Verification is a complex algorithmic task, requiring large amounts of computing resources. One approach to reduce the resource consumption is to reuse information from previous verification runs. This paper gives an overview of three techniques for such information reuse. Conditional model checking outputs a condition that describes the state space that was successfully verified, and accepts as input a condition that instructs the model checker which parts of the system should be verified; thus, later verification runs can use the output condition of previous runs in order to not verify again parts of the state space that were already verified. Precision reuse is a technique to use intermediate results from previous verification runs to accelerate further verification runs of the system; information about the level of abstraction in the abstract model can be reused in later verification runs. Typical model checkers provide an error path through the system as witness for having proved that a system violates a property, and a few model checkers provide some kind of proof certificate as a witness for the correctness of the system; these witnesses should be such that the verifiers can read them and —with less computational effort— (re-) verify that the witness is valid.

1 Introduction

Algorithms for automatic verification require large amounts of computing resources [2, 18]. Furthermore, one verification run of a single verification tool is often not sufficient to completely solve practical verification problems. The verifier might fail to give an answer for various reasons, for example due to the lack of resources (time and memory), an architectural weakness or a missing feature, or simply due to a bug in the verifier. In such cases, the verification process would often be continued on a more powerful engine or using a different verification approach. Sometimes automatic verifiers report wrong answers, and thus, in safety-critical applications it might be desired to rely not only on a single tool, but instead repeat the verification using several other verifiers in order to increase the confidence in the result. A verification run might also be repeated because it was run by an untrusted third party, and the result needs to be re-checked. Some systems consist of many connected components that are verified independently (compositional verification), or of a series of similar products that differ in the set of features that they contain (product-line verification) [1].

When a system is developed, it is desired to detect specification violations soon after they are introduced in order to support early bug fixing; thus, verification should be applied on each new version of the system (regression verification) [21, 30], and even after each single change. This requires a large number of verification runs and enormous computing resources. Regression checking is state-of-the-art in testing, and regression test selection is a well-known and established technique to reduce the number of tests [27]. Verification tools themselves are also under development, and a regression-checking test suite consisting of many verification tasks with known verification result [4] can be used to detect new bugs in current versions of the verifier.

In all of the above-mentioned verification tasks it would be possible and beneficial to store information from previous verification runs to reduce the computational effort, or to increase the quality of the verification result. More research projects are necessary to provide solutions for more reuse of (intermediate) verification results, and for making the existing verification technology more successful in industrial applications.

We identified three categories in which information from a previous verification run should be used in order to spare computational effort that would otherwise be necessary: (1) the use of partial results of verification runs that were not able to completely verify the system; (2) the reuse of auxiliary information that was computed during previous verification runs in order to speed up later verification runs; (3) the use of witnesses for verifying the correctness of previous results.

For each of these categories we present one example from software verification and illustrate the effectiveness of the approach by some experimental evaluation. First, conditional model checking [8] is an approach in which a verifier takes as input a condition that specifies which parts of the program should be verified, and produces an output condition that specifies which parts of the state space were successfully verified. The output condition of one verification run can be used as the input condition of a subsequent run such that the latter can skip the already-verified parts of the state space and focus on the remaining state space. Second, many approaches that are based on CEGAR [17] use some form of *precision* that specifies the level of abstraction of the abstract model that gets constructed for the analysis of the system (e.g., predicate abstraction [3,7,19,20]). This information about the precision can be dumped after a verification run and read in before starting another run (e.g., [13]), reducing the verification time of the latter run because the precision is already computed and many refinement steps are automatically omitted. Third, model checkers typically provide a counterexample (an error path) if a violation of the specification is found in the system, in order to help the user identifying and eliminating the bug. Such counterexamples can also be used —if exported in a machine-readable format— for (re-)verifying if the result of the model checker is (still) correct.

Related Work. We restrict our discussion of related work to automated software verification. Conditional model checking [8] allows to start the overall verification process using one verifier (depending on the abilities of the verifier, the result might be partial), and later use another verifier to further increase the verification coverage, i.e., check the remaining state space. For example, if model checkers are not able to verify certain properties of the system, (guided) testing tools can be used in a second step to increase confidence of correctness for the remaining, not yet verified parts of the state space [16].

Reusing information from a successful verification run for previous versions of a modified system is the basis of many approaches for regression verification [30]. Different forms of information have been proposed for reuse: state-space graphs [23,24,33], constraint-solving results [31,34], function summaries [29], and abstraction precisions [13]. Some of the data can become quite large compared to the system under investigation, and in most cases there needs to be a validation check on whether it is sound to reuse the information (i.e., whether the information still applies to a new version of the system). Precisions are concise and can be reused by the same algorithm that produces them, without a separate validation step.

Most state-of-the-art model checkers produce a counterexample for inspection by the user if the system violates the property, in order to guide the user in the defect-identification process. However, only a few verifiers support witnesses for verification runs showing that the property holds: more verifiers should provide witnesses for correctness. Well-known forms of witnesses for the correctness of a program are proof-carrying code [26], program invariants [22], and abstract reachability graphs [23]. A program for which a safety proof has been found can also be transformed into a new program that is substantially easier to re-verify [32], although verifying the transformed program does not guarantee that the original program is correct.

Experimental Setup. In order to show that reusing verification results is beneficial in many cases, we perform a series of experiments using the open-source software-verification framework CPAchecker[1]. We use revision 7952 from the trunk of the project's SVN repository. CPAchecker integrates many successful verification approaches. In particular, we use its predicate analysis [11] and its explicit-value analysis [12]. Both are based on CEGAR and lazy abstraction.

The benchmark set that we use in this paper consists of the C programs from the 2nd Competition on Software Verification[2] [5], except for the categories "Concurrency" and "Memory Safety", for which CPAchecker has no support. Thus, our benchmark set contains a total of 2250 C programs, 480 of which contain a known specification violation.

We use machines with an Intel Core i7-2600 3.4 GHz quad-core CPU and 32 GB of RAM, allowing the verifier to use two cores (plus two hyper-threading cores) and 15 GB of RAM. The time limit is set to 15 minutes of CPU time. We run two independent instances of the verifier in parallel on each machine, in order to speed up the benchmarking. The operating system of the machines is Ubuntu 12.04 with Linux 3.2 as kernel and OpenJDK 1.7 as Java virtual machine.

[1] `http://cpachecker.sosy-lab.org`
[2] `http://sv-comp.sosy-lab.org`

The CPU time is measured and reported in seconds with two significant digits. The memory consumption is measured including the Java VM that CPAchecker uses as well as the memory that all other components of the verification process (e.g., the SMT solver) need and is given in megabyte with two significant digits.

We present our results using scatter plots that compare a configuration without information reuse versus a configuration with information reuse. Each data point in such a plot represents the performance results of one verification task, where the x-value reports the verification time that is needed in the initial run, and the y-value reports the verification time that is needed in the second run, in which some information from the first run was reused. Thus, data points in the lower-right triangle (with the x-value greater than the y-value) show a speedup through information reuse, with the performance benefit increasing with the distance of the data point from the diagonal. Instances that cannot be verified due to a timeout of the verifier are shown with a time of 900 s and are drawn at the right or top of the plots.

2 Conditional Model Checking

In traditional model checking, the outcome of a verification run is either "safe" or "unsafe". However, it may also happen that a model checker fails and produces no result at all, for example due to resource exhaustion. In such cases, the computational effort that was invested is lost. Conditional model checking [8] redefines model checking in order to solve this problem. A conditional model checker gives as output a *condition* Ψ that states under which condition the analyzed program satisfies the given specification. This condition is produced even in case of a failure, and thus, the consumed resources are not wasted because every run produces some useful result. The previous outcome "safe" translates to $\Psi = true$, and the outcome "unsafe" translates to $\Psi = false$, however, the condition allows for more flexible outcomes. For example, in case of a timeout, a conditional model checker would summarize the already verified part of the state space in the condition, stating that the program is safe, as long as its execution stays within this part. In case of an unsound analysis like bounded model checking with a fixed loop bound, or an algorithm with an incomplete pointer-alias analysis, these assumptions for program safety would also be explicitly given in the output condition, e.g., the program is safe under the assumption "pointers `p` and `q` are not aliased".

Furthermore, a conditional model checker also takes as input a condition that specifies parts of the state space that are already verified, i.e., which the model checker can omit and should not verify again. This can be used to restrict the analysis, e.g., to at most k loop unrollings (well-known as bounded model checking [14]), to paths not longer than n states, or to some maximum amount of time or memory.

Conditional model checking makes it possible to combine two (or more) verifiers and leverage the power of both. Figure 2 illustrates two example combinations, sequential combination with information passing and combination by

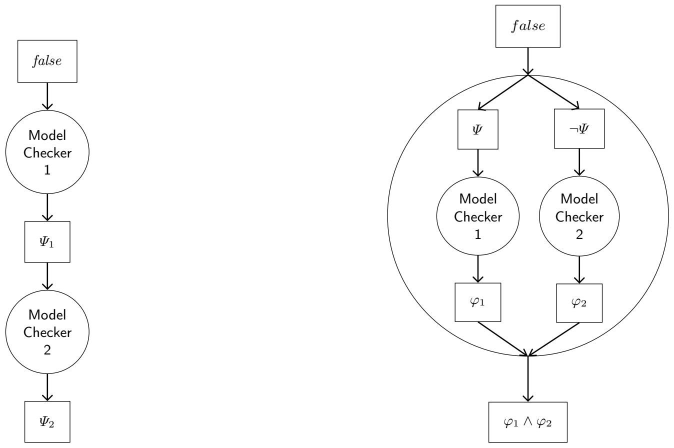


Fig. 1. Combination strategies using conditional model checkers; left: sequential combination with information passing; right: combination by partitioning (compositional verification)

partitioning; for more application examples, we refer the reader to the full article [8]. In contrast to previous combinations of different techniques (e.g., reduced product for combining different abstract domains [9, 15]), the techniques that are combined using conditional model checking can be implemented in different tools, can run on different platforms, even at different locations, or in the cloud; because the interaction and information exchange is realized via implementation-independent conditions.

Sequential Combination with Information Passing. Conditional model checking supports a sequential combination of two verifiers, such that the output condition of the first verifier (describing the successfully verified state space) can be used as input condition for the second verifier. This way, the second verifier will not attempt to verify the state space that was already proven safe by the first verifier. The left part of Fig. 2 illustrates how information can be passed from the first to the second model checker through condition Ψ_1; per default, the first model checker starts with *false* as input condition, i.e., nothing is already verified. The condition Ψ_1 represents the state space that the first model checker was able to verify. The second model checker starts with Ψ_1 as input and tries to verify the state space outside of Ψ_1, i.e., tries to weaken the condition. If the second model checker terminates with output condition $\Psi_2 = true$, then the sequential combination was successful in completely solving the verification problem. If already the first model checker returns $\Psi_1 = true$, then the second model checker has nothing to do; otherwise, the sequential combination is reusing information from the first verification run in the second verification run, making the analysis more powerful than any of them alone.

It is well known that different verification techniques have different strengths and weaknesses on different kinds of programs; the same applies to program parts.

For example, consider a program that contains loops with many iterations as well as non-linear arithmetic. An explicit-state analysis might fail on the loops due to resource exhaustion, whereas a predicate-based analysis might not be able to reason about non-linear arithmetic, and thus, none of the two techniques is able to verify the program on its own. Given an implementation of each analysis as conditional model checker, and a setup that reuses the output condition of one as the input condition for the other, verification of such a program becomes possible. One could run the (conditional) explicit-value analysis first, specifying a maximum path length as input condition. Thus the analysis would not waste all available resources on endlessly unwinding loops, but instead verify the rest of the program, and summarize the results in the output condition. If the subsequent run of the predicate analysis gets this information as input condition, it can focus on the still-missing parts of the state space (the loops), and skip the rest (which in this case, the predicate analysis would not be able to verify due to the non-linear arithmetic). Thus, the complete analysis might prove the program safe, although the same sequential combination without information reuse would not be able to verify the program.

Combination by Partitioning. Conditional model checking also supports compositional verification, which can be set up as a combination where the state space is partitioned into two partitions and two verifiers are started each with an input condition that represents its (negation of the) partition. This way, each verifier concentrates on different aspects of the verification task. If both verifiers succeed to relax the condition to *true*, then the verification task is completely solved. Otherwise, the output condition $\phi_1 \wedge \phi_2$ represents the state space that was successfully verified. This concept allows a convenient construction of compositional verification strategies. In this paper, in which we focus on reuse of verification results, we now concentrate on experiments with the sequential composition.

Experimental Evaluation. We refer to previous experimental results from 2012 [8] to give evidence that conditional model checking, and the combination of verifiers that it makes possible, can verify more programs in less time. For those experiments, we used a benchmark set that consists of 81 programs created from the programs in the categories “SystemC” and “DeviceDrivers64” (two categories that were considered particularly hard) of the Competition on Software Verification 2012 (SVCOMP’12) [4].

As an example for conditional model checking we show the results for a configuration that combines two verifiers sequentially with information passing between the verifiers. The first verifier that is used is an explicit-value analysis that is quite fast for some programs but inefficient for other, more complex programs due to state-space explosion. This analysis is configured to stop itself after at most 100 s. If it terminates without a complete result “safe” or “unsafe” (due to the timeout, or due to imprecision), it dumps a summary of the successfully verified state space as an output condition. The second verifier, which uses a powerful predicate analysis based on CEGAR, lazy abstraction, and adjustable-block encoding [11], continues the verification for the remaining time up to the

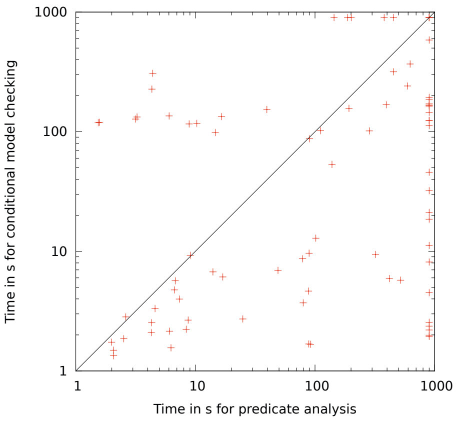


Fig. 2. Scatter plot comparing the verification time of a predicate analysis with the verification time of a conditional-model-checking configuration that uses both an explicit-value analysis and a predicate analysis

global time limit of 900 s. This analysis takes the output condition that was produced by the first verifier as input condition such that it will ignore the already verified state space and focus on the remaining parts. We compare this instance of conditional model checking against a stand-alone predicate analysis, in the configuration that was submitted to SVCOMP'12 [25].

The stand-alone predicate analysis is able to solve 58 of the 81 verification tasks in 31 000 s. The configuration based on conditional model checking instead verified 75 programs in only 14 000 s. Figure 2 presents the verification times for both configurations for 78 out of the 81 benchmark verification tasks (excluding 3 cases where one verifier ran out of memory and aborted prematurely). The majority of the data points is positioned in the lower-right triangle, which shows the performance advantage of conditional model checking. In some cases, the verification time for conditional model checking is just over 100 s and the predicate analysis alone needs only a few seconds. These are verification tasks for which the explicit-value analysis, which is started first in our setup of conditional model checking, is not able to solve the program in its time limit of 100 s, and the predicate analysis that is started subsequently verifies the programs in a short time. Note that there is a significant amount of data points to the right-most area of the plot: these are the verification tasks on which the predicate analysis alone times out. Some of these programs even take more than 100 s when verified with conditional model checking, which means that they were successfully verified by the predicate analysis after the explicit-value analysis terminated, although the predicate analysis alone could not successfully verify them. The verification of these programs is only possible by information reuse, that is, by restricting the predicate analysis to the state space that the explicit-value analysis could not successfully verify. A simple sequential combination of both analyses without information passing would not have been able to verify those programs.

3 Precision Reuse

There are many applications for re-verifying a program that was already verified. Common to all these cases is the fact that information from previous verification runs for the same program would in principle be available, and could be used to speedup subsequent verification runs. Thus, it seems worthwhile to save such information in a machine-readable way after each verification run for the program, for later reuse.

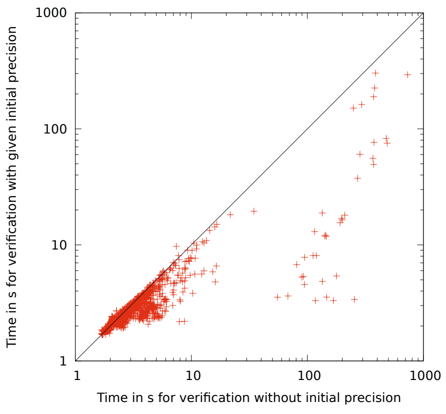
Several successful software-verifiers are based on CEGAR, and continuously refine an abstract model of the program to be analyzed, until the model is strong enough to prove safety or find a violation of the property. The *precision* (level of abstraction) that is used for constructing and verifying the abstract model is crucial information for the success of such CEGAR-based analyses, and discovering an appropriate precision is usually one of the most expensive tasks of the verifier (possibly involving a large number of refinement steps). However, given the precision as input, the verifier can immediately construct an appropriate abstract model and verify the abstract model without further refinements. Thus, such precisions are suited for being reused in subsequent verification runs, as was shown in previous work for the application of this concept to regression verification [13]. For example, predicate analysis with CEGAR and lazy abstraction (e.g., [7]) is a well-known analysis that uses precisions. In this case, the precision contains the set of predicates over program variables that are tracked by the analysis, and (in case of lazy abstraction) the program locations at which the predicates are relevant. A precision can also be used for explicit-value model checking [12], in which case the precision stores the program variables that are relevant for the verification; all other program variables should be abstracted away by the verifier. Similar precisions can be used for analyses based on other abstract domains, such as intervals or octagons. Precisions are usually much smaller than the program itself, and can be easily dumped after the verification run.

If a precision for a given program is present from a previous verification run, it can easily be used as the initial precision of a subsequent verification run, instead of the usual (coarse) initial precision. Thus, no refinements are necessary anymore (given that the program was not changed). In contrast to other approaches like proof checking, where a separate algorithm is needed for verifying the proof, precision reuse does not require a new algorithmic setup: the same analysis and algorithm that produce the precision in a first verification run are the components that use the precision in a subsequent verification run. Furthermore, if the provided precision does not fit to the program (for example because the program was changed, or the user provided a wrong input file), there is no risk of incorrect verification results. Instead, the verifier will simply detect that the abstract model is not strong enough to verify the given property by finding spurious counterexamples, and will use refinements to strengthen the abstract model as it would in a verification run without a given input precision.

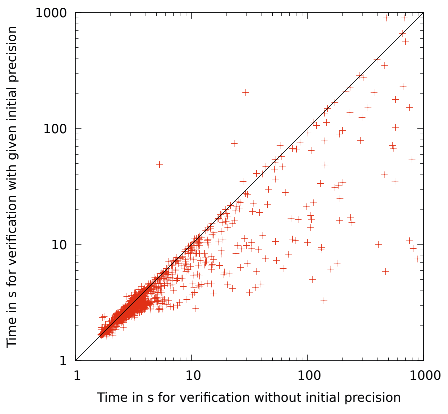
Experimental Evaluation. Both the predicate analysis and the explicit-value analysis of CPAchecker are based on CEGAR and use a precision to define the

Table 1. Results for precision reuse

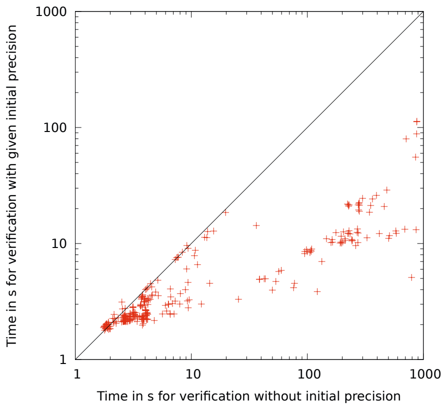
Analysis	Programs	Without precision reuse				With precision reuse			
		Solved Tasks	CPU Time	Memory Avg.	Memory Max.	Solved Tasks	CPU Time	Memory Avg.	Memory Max.
Explicit-Value	Safe	1529	13000	270	9600	1529	6100	170	8200
	Unsafe	298	23000	1400	8100	298	2000	320	3200
Predicate	Safe	1518	27000	280	13000	1516	13000	210	12500
	Unsafe	422	16000	480	8700	420	11000	360	8600



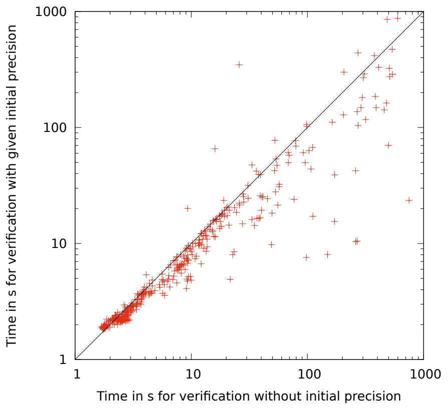
(a) Explicit analysis (safe programs)



(b) Predicate analysis (safe programs)



(c) Explicit analysis (unsafe programs)



(d) Predicate analysis (unsafe programs)

Fig. 3. Scatter plots comparing the verification time without input precision versus with precision reuse

level of abstraction. We used the existing implementation for writing precisions to disk after each verification run and for reading an initial precision from disk before the verification. We experimented with precision reuse for all 2 250 programs of the benchmark set described in Sect. 1. A summary of all results can be found in Table 1. Detailed results are provided on the supplementary web page [3].

Out of the 1 770 programs that are known to be safe, the explicit-value analysis of CPAchecker successfully verified 1 529 instances in 13 000 s, using 270 MB of memory on average. Out of the 480 unsafe programs, 298 were verified in 23 000 s, using 1 400 MB of memory on average. If we reuse the precisions that were produced in these runs as initial precision in a second run, the verification takes only 6 100 s, i.e., less than half of the time for the safe programs, and 2 000 s, i.e., less than 10 %, for the unsafe programs. The memory consumption is also considerably lower if reusing a given precision, dropping (for the unsafe programs) from 1400 MB on average to 320 MB.

The predicate analysis could successfully verify 1 518 out of 1 770 safe programs in 27 000 s. With the precisions reused, 1 516 programs can be verified in only 13 000 s. There are two programs that were verified in the initial run in 480 s and 680 s, respectively, but CPAchecker reached the timeout of 900 s when the precision was given as input. Also there are three programs for which the verification with precision reuse needs significantly more time (factor 3 to 10). For all five programs mentioned above, there was only a small number of refinements (1 to 7) in the initial run, and only less than 4 s was spent on these refinements per program (mostly even around only 0.5 s). This means that the potential benefit was already small for these programs. Furthermore, CPAchecker uses lazy abstraction and thus may have used different precisions on different paths of the programs. Our implementation of precision reuse, however, assigns the same precision on all paths of the program, leading to a possibly stronger and thus more expensive abstract model. This is not a general drawback of precision reuse.

The results for the unsafe programs are similar. The predicate analysis finds a counterexample for 422 out of 480 unsafe programs in 16 000 s, and using the generated precisions as input it finds 420 counterexamples in 11 000 s. Again, two programs cannot be verified when a precision is reused, and there are a few cases for which the necessary verification time is higher. For unsafe programs, there are also other factors that influence the results. For example, depending on the order in which the control-flow automaton is traversed, the analysis might find the first counterexample sooner or later, and with more or less refinements, thus with a different potential performance advantage by precision reuse.

Figure 3 illustrates the results using four scatter plots, one for each CPAchecker configuration, and for the safe and the unsafe programs. Cases in which the verifier timed out in the initial run and thus produced no reusable precision are omitted. The graphs show that precision reuse is beneficial, because the vast majority of data points are located in the lower-right triangle. For the explicit-value analysis, there is no verification task for which the run time is significantly increased by precision reuse.

[3] `http://www.sosy-lab.org/~dbeyer/cpa-reuse-gen/`

4 Verification Witnesses and Their Re-verification

It is common that model checkers produce a counterexample if the system violated the property, as witness of the verification result. The main purpose of the counterexample is to convince the user of the verification result and to guide the user in the defect-identification process. Below we argue that it is necessary to (1) produce a counterexample in a machine-readable format, such that the counterexample can be re-verified later, and to (2) analyze the counterexample for feasibility not in isolation, but in relation to the program to be analyzed.

It would also be desirable to produce witnesses for verification runs that prove that the property holds. It seems to be an open research question to achieve this, perhaps because a witness for correctness can have a size exponential in the size of the input program. There are important research results available on witnesses for the correctness of a program, for example, proof-carrying code [26], program invariants [22], and abstract reachability graphs [23]. Unfortunately, it did not yet become state-of-the-art to support those techniques in verification tools. Hopefully, since tools for software verification became more mature in the last years, as witnessed by the competition on software verification [5], there will be a need for certification of verification claims. That is, in the future, it will not be sufficient to report a verification answer ("safe" or "unsafe"), but one has to support the claim by a verification witness (proof certificate).

Re-verification of Counterexamples. The re-verification of previous verification results can be supported by intermediate results, as outlined in the previous sections, but also by providing witnesses for the verification result. We now consider the re-verification of verification results where a violation of the property is reported and a counterexample is produced. It seems obvious that verifying if only a single given path out of the program violates the specification is more efficient than verifying the complete program and finding a specification-violating path in it. Our experiments support this claim with encouraging numbers, showing that the benefit is indeed present, even if the counterexample is re-verified against the original program. There are two important properties that the witness-based re-verification has to fulfill: the use of machine-readable counterexamples and the re-verification against the original program.

Machine-readable counterexamples. First, we need the verifier to dump information about the found counterexample of an unsafe program in a *machine-readable* format, which can later be reused in a re-verification run to restrict the verification process to this single path (e.g., by giving the negation of the counterexample as input to a conditional model checker). One possibility would be to dump the source code of a new program that corresponds to a single counterexample of the original program. This new program would be free of loops and branches, and thus hopefully easy to verify. However, in the case where the goal of re-verifying a counterexample is increased confidence, this is not a good idea. If the verifier that is used in the first verification run is imprecise and reports an infeasible counterexample, it might generate a witness program that does contain a specification violation, but does not correspond to an actual path of the original

```
CONTROL AUTOMATON PathGuidingAutomaton

INITIAL STATE s0;

STATE USEFIRST s0:
  // match declaration statement of program and goto state s1
  MATCH "int x;" -> GOTO s1;
  // match all other statements and stop exploration of path
  TRUE           -> STOP;

STATE USEFIRST s1:
  // match assume statement of program and signal specification violation
  MATCH "[x==0]" -> ERROR;
  // match all other statements and stop exploration of path
  TRUE           -> STOP;

END AUTOMATON
```

Fig. 4. Example automaton for guiding the verifier along a certain path (written in CPAchecker's specification language), which can be used for re-playing a previously reported counterexample on the original program

program. In this case, the second verifier would correctly claim that the witness program is indeed unsafe, leading the user to an incorrect conclusion about the correctness of the original program.

"Re-playing" Counterexamples. Second, counterexamples should be re-verified against the original program, not in isolation. This strategy is motivated by verification results delivered from untrusted verification engines, the need to re-verify slightly changed programs (regression verification), and excluding spurious counterexamples that were reported by imprecise verification tools. For the implementation of this strategy —using the original program as input for the re-verification run— we propose to use a simple language for automata that guide the verifier along a certain path, in order to have the verifier exactly re-play the previously found counterexample. The automaton needs to be able to match operations of the program (possibly by textual matching, or by line numbers), to guide the verification, preventing the exploration of unrelated paths, and to specify a certain state of the program as a target state whose reachability should be checked by the verifier. Previous work on specification languages based on automata can be used to implement this strategy (e.g., [6, 10, 28]).

We can use the automaton language that the verifier CPAchecker accepts as specification format for counterexamples without any changes. An example for such an automaton is given in Fig. 4. An automaton for guiding the verifier along a single path in a program consists of a set of states, where each state has exactly one edge that matches a single program operation and leads to the successor state. For all other program operations that cannot be matched, the automaton instructs the verifier to stop exploring the path along that program operation. At the end of this chain of states, the automaton switches to a special

Table 2. Results for re-verification of counterexamples

Analysis	Initial verification				Re-verification			
	Solved Tasks	CPU Time	Memory Avg.	Memory Max.	Solved Tasks	CPU Time	Memory Avg.	Memory Max.
Explicit-Value	299	24000	1400	8400	299	870	140	890
Predicate	422	18000	490	8900	422	1300	120	590

error state, which informs the verifier that the corresponding program state is a specification violation. If the verifier reaches this state, then it reports the program as unsafe.

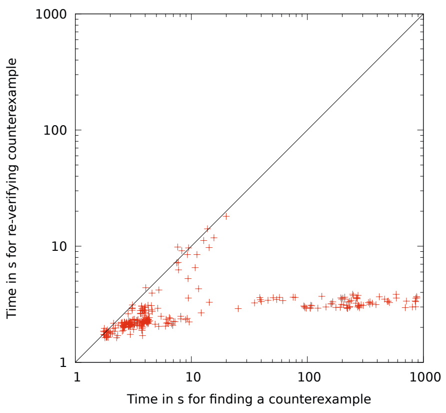
Such an automaton that matches program operations along a counterexample path is easy to produce for all kinds of analyses that are able to reproduce a single finite path through the control-flow of the verified program as a representation of a counterexample. This includes analysis approaches based on creating abstract reachability graphs (which are unrollings of the control flow), but also other analyses like bounded model checking, if some information about the structure of the control flow is encoded in the generated formula and a path is reconstructed using the information from a model for the program formula.

The automaton is also easy to use as input for the re-verification step, if the verifier is based on traversing the control-flow of the program. In this case, whenever the verifier follows a control-flow edge, it would also execute one edge of the automaton and act accordingly (i.e., continue or stop exploring this path). Again, this strategy is applicable to verifiers based on abstract reachability graphs, but also to others. For bounded model checking, this can be implemented in the first phase where the program is unrolled and a single formula is created representing the program. With such an automaton, the unrolling would be restricted and the generated formula represents only that single path, which could then be verified by checking the formula for satisfiability as usual. The complexity of both generating and using the automaton is linear in the length of the counterexample.

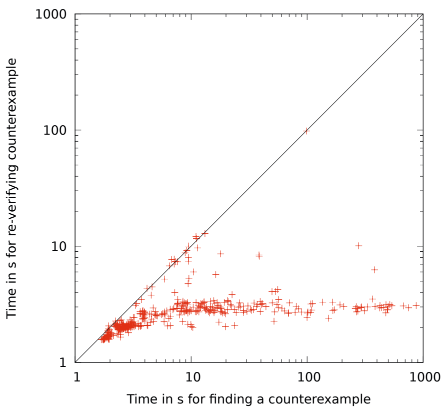
Experimental Evaluation. To support experiments with re-verification of counterexamples, we implemented the export of a counterexample as automaton in CPAchecker's specification language. Our implementation in the CPAchecker framework can be used with all available configurable program analyses that are based on abstract reachability graphs and is available via the project's SVN repository.

We experimented again with the explicit-value analysis and the predicate analysis. A summary of the results is presented in Table 2. Detailed results are provided on the supplementary web page[4]. The explicit-value analysis of CPAchecker finds the bug in 299 out of the 480 unsafe programs from our benchmark set (for the remainder, it fails or runs into a timeout). The produced counterexample automaton can be used as verification witness, e.g., in a second

[4] `http://www.sosy-lab.org/~dbeyer/cpa-reuse-gen/`



(a) Explicit analysis



(b) Predicate analysis

Fig. 5. Scatter plots comparing the run time for finding a counterexample in the complete program vs. re-verifying a given counterexample

verification run of the original program in order to explicitly verify only this single path. The re-verification confirms the counterexample for all verification tasks. The run time for finding the counterexamples in the first run was 24 000 s for the 299 programs. The run time for re-verifying the produced counterexamples was only 870 s, i.e., less than 4 %. The average memory consumption was 1 400 MB for the initial runs, and 140 MB for the re-verification runs.

CPAchecker's predicate analysis could find a counterexample for 422 programs in 18 000 s. The re-verification of these counterexamples took only 1 300 s. There was only one verification task for which the re-verification took longer than 13 s. For 40 verification tasks, the initial verification run to identify a counterexample took longer than 100 s.

The maximum memory consumption per analyzed program, i.e., the amount of memory that the machine needs to have available, is also lower for re-verification. For the initial verification runs, the maximum memory consumption was 8 400 MB (explicit-value analysis) and 8 900 MB (predicate analysis). During the re-verification of the counterexamples, the maximum memory consumption was 890 MB and 590 MB, respectively. This means the following: while for finding the bugs in the complete programs, the machine needs to be powerful (more than 8 GB of RAM are still not common for developer machines), the re-verification can be performed on practically any available machine (even machines older than 8 years and small netbooks tend to have at least 1 GB).

Scatter plots for the results of all successfully verified programs are shown in Fig. 5. The results are interesting: the verification time for re-verifying a counterexample is less than 4 s for most of the programs, regardless of the verification time that was needed for finding the counterexample in the original program. There are no verification tasks for which the verification time for re-verification significantly exceeds the run time for the initial verification run.

It is an important insight to have confirmed that the re-verification can be performed on a much less powerful verification engine, and thus, is significantly less expensive overall. This justifies the use of untrusted computing engines for the verification process: it is reasonably inexpensive to confirm the correctness of verification results that arrived with status "unreliable".

5 Conclusion

We have shown that the reuse of verification results from previous verification runs can save significant amounts of resources (time and memory). As example applications, we explained three different strategies for reusing verification results: conditional model checking, precision reuse, and verification witnesses. We illustrated the benefits of reusing verification results by reporting on experimental results. Reusing verification results from previous verification attempts does not only improve the performance, but sometimes also the effectiveness, i.e., more verification tasks can be solved. Systems that are currently still too complex to be verified by one single verifier could be verified by combination and information passing between verification runs. More such techniques need to be developed and used in the future, in order to apply automatic verification to large-scale industrial systems.

Important future work, in order to make information reuse practically applicable, includes research on defining standardized formats. Such standard formats are the key for combining different verification tools and for reusing (partial) verification results across different verification approaches. For example, a common format for verification witnesses (proof certificates as well as counterexamples) would increase the adoption of verification technology by verification engineers in practice, by providing an easy way for re-verifying results and integrating verification within development tool chains. Besides the conditions of conditional model checking and the witnesses, we demonstrated that intermediate results such as precisions have a lot of potential for reuse. More research is necessary to investigate which information is to be saved and reused.

References

1. Apel, S., von Rhein, A., Wendler, P., Größlinger, A., Beyer, D.: Strategies for product-line verification: Case studies and experiments. In: ICSE 2013. IEEE (2013)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008)
3. Ball, T., Rajamani, S.K.: The Slam project: Debugging system software via static analysis. In: POPL 2002, pp. 1–3. ACM (2002)
4. Beyer, D.: Competition on software verification (SV-COMP). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012)
5. Beyer, D.: Second competition on software verification. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 594–609. Springer, Heidelberg (2013)

6. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The Blast query language for software verification. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 2–18. Springer, Heidelberg (2004)
7. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007)
8. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: FSE 2012. ACM (2012)
9. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
10. Beyer, D., Holzer, A., Tautschnig, M., Veith, H.: Information reuse for multi-goal reachability analyses. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 472–491. Springer, Heidelberg (2013)
11. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: FMCAD 2010, pp. 189–197. FMCAD (2010)
12. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Cortellessa, V., Varró, D. (eds.) FASE 2013. LNCS, vol. 7793, pp. 146–162. Springer, Heidelberg (2013)
13. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Reusing precisions for efficient regression verification. Technical Report MIP-1302, University of Passau (2013)
14. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
15. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI 2003, pp. 196–207. ACM (2003)
16. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 132–146. Springer, Heidelberg (2012)
17. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
18. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT (1999)
19. Clarke, E.M., Kröning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
20. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
21. Hardin, R.H., Kurshan, R.P., McMillan, K.L., Reeds, J.A., Sloane, N.J.A.: Efficient regression verification. In: WODES 1996, pp. 147–150 (1996)
22. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 526–538. Springer, Heidelberg (2002)
23. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 332–358. Springer, Heidelberg (2004)
24. Lauterburg, S., Sobeih, A., Marinov, D., Viswanathan, M.: Incremental state-space exploration for programs with dynamically allocated data. In: ICSE 2008, pp. 291–300. ACM (2008)

25. Löwe, S., Wendler, P.: CPAchecker with adjustable predicate analysis (competition contribution). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 528–530. Springer, Heidelberg (2012)
26. Necula, G.C.: Proof carrying code. In: POPL 1997, pp. 106–119. ACM (1997)
27. Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. IEEE Trans. Softw. Eng. 22(8), 529–551 (1996)
28. Šerý, O.: Enhanced property specification and verification in Blast. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 456–469. Springer, Heidelberg (2009)
29. Šerý, O., Fedyukovich, G., Sharygina, N.: Incremental upgrade checking by means of interpolation-based function summaries. In: FMCAD 2012, pp. 114–121. FMCAD (2012)
30. Sokolsky, O.V., Smolka, S.A.: Incremental model checking in the modal μ-calculus. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 351–363. Springer, Heidelberg (1994)
31. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: Reducing, reusing, and recycling constraints in program analysis. In: FSE 2012. ACM (2012)
32. Wonisch, D., Schremmer, A., Wehrheim, H.: Programs from proofs: A PCC alternative. In: CAV 2013, LNCS. Springer (2013)
33. Yang, G., Dwyer, M.B., Rothermel, G.: Regression model checking. In: ICSM 2009, pp. 115–124. IEEE (2009)
34. Yang, G., Păsăreanu, C.S., Khurshid, S.: Memoized symbolic execution. In: ISSTA 2012, pp. 144–154. ACM (2012)