Conditional Model Checking: A Technique to Pass Information between Verifiers [*‡]

Dirk Beyer
University of Passau
Germany

Thomas A. Henzinger
IST Austria
Austria

M. Erkan Keremoglu
Simon Fraser University
Canada

Philipp Wendler
University of Passau
Germany

ABSTRACT

Software model checking, as an undecidable problem, has three possible outcomes: (1) the program satisfies the specification, (2) the program does not satisfy the specification, and (3) the model checker fails. The third outcome usually manifests itself in a space-out, time-out, or one component of the verification tool giving up; in all of these failing cases, significant computation is performed by the verification tool before the failure, but no result is reported. We propose to reformulate the model-checking problem as follows, in order to have the verification tool report a summary of the performed work even in case of failure: given a program and a specification, the model checker returns a condition Ψ —usually a state predicate— such that the program satisfies the specification under the condition Ψ —that is, as long as the program does not leave the states in which Ψ is satisfied. In our experiments, we investigated as one major application of conditional model checking the sequential combination of model checkers with information passing. We give the condition that one model checker produces, as input to a second conditional model checker, such that the verification problem for the second is restricted to the part of the state space that is not covered by the condition, i.e., the second model checker works on the problems that the first model checker could not solve. Our experiments demonstrate that repeated application of conditional model checkers, passing information from one model checker to the next, can significantly improve the verification results and performance, i.e., we can now verify programs that we could not verify before.

Categories and Subject Descriptors: D.2.4 [*Software Engineering*]: Software/Program Verification F.3.1 [*Logics and Meanings of Programs*]: Specifying, Verifying, Reasoning about Programs

Keywords: Formal Verification, Model Checking, Program Analysis, Sequential Combination, Coverage, Testing

[*]This research was supported by the Canadian NSERC grant RGPIN 341819-07, the ERC Advanced Grant QUAREM, and the Austrian Science Fund NFN RiSE.
[‡]A preliminary version of this article appeared as Technical Report MIP-1107, University of Passau, in 2011 [6].

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

1. INTRODUCTION

Model checking is an automatic search-based procedure that exhaustively verifies whether a given model (e.g., labeled transition system) satisfies a given specification (e.g., temporal-logic formula) [12, 33]. Since model checking of software is an undecidable problem, there are three possible outcomes of the analysis process: (1) the program satisfies the specification, (2) the program does not satisfy the specification, and (3) the model checker fails. The first outcome can be obtained by the model checker if the abstract model that was computed for the program is sufficient to prove the program correct under the given specification. This outcome can be accompanied by a proof certificate [23]. The second outcome can be obtained by the model checker if an abstract counterexample path is found and can be proven feasible, i.e., a bug that can actually occur in the program. This outcome is usually accompanied by the violating program part in the form of program source code, and sometimes test input to reproduce the error at run-time [4]. The third outcome usually occurs if the model checker runs out of resources (memory exhausted, time-out) or if one of the components in the verification tool gives up. In all of these failing cases, significant computation is performed by the verification tool before the failure. But since no useful result is reported, the spent resources are wasted.

Our goal is to capture the results of the model checker, and take those results as input for further verification efforts. For example, knowing the state space that was already proved safe by the first tool, a second tool can focus straight on parts of the state-space that are not yet verified. We propose the approach of *conditional model checking*. The goal of conditional model checking is to maximize the outcome of a model-checking run under certain conditions, e.g., a given set of resources. We reformulate the model-checking problem as follows: Given a program, a specification, and an input condition, the model checker returns a condition Ψ —usually a state predicate— such that the program satisfies the specification under the condition Ψ —that is, as long as the program does not leave the states in which Ψ is satisfied. The condition Ψ represents the state space that has been verified, thus, we are interested in model checkers that return conditions Ψ that are as weak as possible.

The outcomes of a model-checking run can be translated to conditions in the following way: Previous outcome (1) corresponds to the condition $\Psi = \mathit{true}$. That is, if the model checker returns *true* as condition, the model checker completely verified the program under no additional conditions. For outcome (2), the model checker does not return *false*,

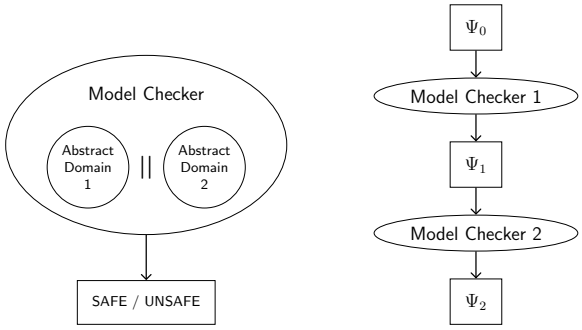

Figure 1: Combination strategies
Left: Parallel combination of two abstract domains; combination must be implemented within the same tool and algorithm; information can be exchanged at any time;
Right: Sequential combination of two analyses with conditional model checking; the techniques can be implemented in different tools, can run on different platforms, even at different locations in the cloud; information can be transferred from the first to the second tool through condition Ψ_1; per default, the first model checker starts with $\Psi_0 = \mathit{false}$.

but can specify program parts that are free of errors, and explicitly exclude the violating parts. For example, consider an invalid program that consists of two branches, one of which is safe and the other violates the specification. The resulting condition for this program would contain all program states that occur in the successfully verified branch and report an error path that violates the specification. In outcome (3), in which the model checker previously failed with "no useful result", the condition Ψ now summarizes the work that has been performed by the model checker before space-out, time-out, or giving up.

If a model checking run was not completely successful, the condition can be taken as input for another model checker that might succeed in proving the program correct or find an error, while it does not need to touch the state space that is already proven correct. Thus, one major application of conditional model checking is a sequential combination of several model checkers where knowledge from an earlier run of the model checker is passed to a later run. Such a combination is strictly more than the sum of individually executing model checkers one after the other in batch mode.

For example, consider a valid program that consists of two branches, one of which is easy to verify by the model checker and the other leads the model checker into an infinite loop. Using conventional model checking, we would not get any feedback from the model checker, because the time-out would cause a failure. Our modified approach of conditional model checking would still be unable to prove that the program satisfies the specification. However, it would —by taking its input condition into account— heuristically detect the hopeless situation and summarize the performed work by reporting that (at least) the first branch has been successfully verified. If complete verification is necessary, then a different verification method or tool may be used to focus on the states that violate the condition.

We performed experiments to investigate the sequential combination of different techniques and tools, and conclude that conditional model checking can significantly improve the performance of the analysis, and that the sequential combination with information passing can effectively check programs that we were not able to verify with conventional model checking or simple sequential combinations.

Contributions. We make the following two contributions:

- *Conditional Model Checking.* We introduce the technique of conditional model checking, which generates a summary of the performed verification work as output —for the user as feedback, or for later tool runs— and accepts as input a condition that specifies which parts of the state space are already verified. We have implemented an extension of an existing model checker in such a way that conditions can be generated as output and accepted as directive input. Such a technique and implementation did previously not exist [6].
- *Sequential Combination of Model Checkers.* We applied conditional model checking to the problem of combining several model-checking techniques sequentially with passing information between them. We performed experiments on combining explicit-state model checking and predicate-based model checking. The goal was to take advantage of the complementary benefits of each technique. The experiments witness a significant improvement in performance and precision. Such verification configurations, where information is passed between two model-checking techniques, were not experimentally evaluated before [6].

Figure 1 explains the difference between parallel (left) and sequential combination (right) of verification techniques. In parallel combination, the two abstract domains analyze and store different aspects of the program. The model checker can combine results and exchange information between the domains at any time (most elegantly in a strengthen step after both post operations). This is, for example, not possible if the techniques require different iteration algorithms or rely on different binaries. In sequential combination with conditions, the two model checkers are not executed independently in batch mode one after the other, in good hope that one succeeds, but connected via a condition. The first tool analyzes the program according to the input condition Ψ_0 (per default, Ψ_0 is *false*). When finishing, the first model checker summarizes its work in Ψ_1, such that the second tool knows what is left to be verified.

Other Applications. Conditional model checking enables the following features that were not possible in conventional model checking before:

- *No Fail without Results.* Every run of the model checker results in a condition that summarizes the achieved results.
- *Maximal Outcome.* Conditional model checking maximizes the outcome for a given set of resources (memory/time limit).
- *Partial Verification.* Conditional model checking can be used to restrict the verification to certain parts of the program, by taking as input a condition that excludes the parts that should not be verified. For example, some parts of the system might be checked via model checking, others via testing, theorem proving, or complementary model checkers.
- *Comparison of Tools.* Given two model-checking tools, we can not only compare the time and memory needed for a given verification task, but we can now compare the quality of the verification results (the weaker the condition, the better).

- *Benchmark Generation.* Conditional model checking can be used to produce hard verification benchmarks. Given a verification task that currently can not be solved, conditional model checking can be instructed to generate excluding conditions for all parts that (currently) can not be model checked, and dump a new program that does not contain the state space that is excluded by the conditions. If started on the new program (which represents the maximal verifiable program fragment), the model checker succeeds and the time to prove this fragment correct can be measured.

- *Guided Test-Case Generation.* The output condition of conditional model checking can be used to guide a tool for test-case generation, if previous verification attempts were not successful. The test generator uses the condition to produce test cases that explore the not-yet-verified state space, but does not produce test cases for the already verified state space. A concrete instance of this application idea was recently described [11]. Such a sequential combination of static analysis and testing is orthogonal to previous, more tight combinations, like SYNERGY/YOGI [22].

Example. Loops introduce challenges for static program analysis. For example, if the abstract domain of predicates with lazy abstraction is used for the analysis, there is the possibility that each step of loop unwinding will add a new predicate and verification will be performed until the entire path is unwound. In some cases that operation might be repeated thousands of times and this will lead the analysis to get stuck in the loop. The example in Fig. 2 presents a case where the analysis might fail to terminate. If the analysis has to fully unwind the loop in this program, the analysis will not terminate in a reasonable amount of time. In that case, the verification outcome would be 'fail'. At line 11, there is another assertion and the analysis would miss the opportunity to investigate this part of the program because it is busy with checking the loop. If the analysis is started with a condition that limits the number of unwindings of a loop to at most k iterations, it will eventually skip the loop and verify the rest of the program. This way it can determine that the assertion at line 11 does not hold and report an error without much effort. If the specification were not violated in the rest of the program, the outcome would be 'safe' under the condition that the program visits the loop entry at most k times (as in bounded model checking).

Model checking with predicate analysis depends on the capabilities of SMT solvers, and thus can only verify conditions that can be expressed in theories that are supported by the integrated SMT solver. This might exclude, for example, properties that depend on non-linear relations between program variables, as shown in the snippet of Fig. 3. Suppose we analyze this program with a conventional predicate analysis. Given a precision that includes the predicate $(i \geq 1000000)$, the model checker will easily be able to prove that the assertion in line 5 can never fail. However, as our predicate analysis is based on linear arithmetics and needs to model the multiplication of program variables as uninterpreted function, the model checker cannot prove that the second assertion in line 11 also always holds. A precise path analysis of the counterexample reveals that the assertion cannot fail, thus, the analysis has to give up.

```
1  void main() {
2    int x = 1;
3    if (nondet_int()) {
4      while (x < 10000) {
5        x++;
6      }
7      assert (x == 10000);
8    } else {
9      x = 0;
10   }
11   assert (x != 0);
12 }
```

Figure 2: Example program with loop

```
1  void main() {
2    if (nondet_int()) {
3      int i;
4      for (i = nondet_int(); i < 1000000; i++);
5      assert(i >= 1000000);
6
7    } else {
8      int x = 5;
9      int y = 6;
10     int r = x * y;
11     assert(r >= x);
12   }
13 }
```

Figure 3: Example with non-linear safety condition

In order to complete the verification of the program in Fig. 3, a second analysis is needed. In this case a good choice for verifying the assertion in line 11 is an explicit-value analysis for integers. If the condition produced by the predicate analysis is used to restrict the analyzed state space, the explicit-value analysis needs to prove only the safety of the second assertion, which it can check efficiently. The final analysis correctly reports that the program is safe. However, note that an explicit-value analysis cannot verify the other assertion (it would have to unroll the loop and branch many times, probably exceeding all available time or memory resources). Thus it is indeed necessary to use the condition from the first run to guide the second analysis.

By using two different model-checking configurations, and by giving detailed information about the verified state space in form of a condition, this example can be proved safe. None of the two configurations is able to verify this alone; both would either fail due to resource exhaustion or terminate without useful results.

Availability of Data and Tools. We implemented conditional model checking using standard components of the open-source verification framework CPACHECKER [8]. All experiments are based on publicly available benchmark programs from the last competition on software verification [3] [1]. Our extension of CPACHECKER is available in the project repository [2]. CPACHECKER is released under the free-software license Apache 2. All benchmark programs that we used in our experimental evaluation, as well as the necessary configurations, scripts, and a ready-to-run version of CPACHECKER are available on the supplementary web page [3].

Related Work. The assume-guarantee paradigm is a well-known principle of verification theory [26]. Conditional model checking (CMC) implements this paradigm: "if the

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

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

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

program fulfills the (generated) assumptions, then the program is guaranteed to satisfy the specification". A verifier should explicitly state under which conditions it guarantees correctness. ESC/Java uses an annotation language for Java to let the user encode conditions for pruning false alarms [20]. Thus, the user can choose a compromise between soundness and efficiency. CMC follows this principle by allowing the model checker to take conditions as input.

Bounding the path length in symbolic execution is a well-known technique [28]. Bounded model checking (BMC) is successful in finding bugs, but is often rejected as verification technique because it is unsound [10]. If the condition of unwinding every loop up to a certain bound is stated explicitly, then it would be a sound conditional model-checking technique. The bounded model checker CBMC does report whether the program contains paths that exceed the given bounds, or whether the program could be verified completely [14]. This can be nicely expressed as a condition, but conditional model checking is more powerful because more information is reported. Nimmer and Ernst efficiently extract program specifications from dynamic analysis [32]. Although the approach is unsound, the generated specifications can effectively cover a large part of the state space. Conditional model checking also has the objective to improve the verification coverage. Conway et al. define a points-to analysis with conditional soundness [15].

Many software-verification tools make implicit assumptions about the program, sacrificing soundness in order to be practically useful. For example, the predicate-abstraction–based tools SLAM [1] and Blast [5] implicitly use the assumption that the program does not contain integer overflows, and model variables as unbounded mathematical integers (cf. [5], page 515). Several verification tools use heuristics similar to our conditions (search strategies, iteration orders, pruning, etc.) in order to find safety violations faster [18, 31]. JPF is a tool that allows arbitrary user-defined search strategies [21]. Conditional model checking makes these heuristics externally visible and adjustable in the form of conditions.

No experimental investigation was done yet on using such conditions for sequentially combining model checkers beyond executing them one after the other in isolation. Our experimental study shows that sequential combination with information passing (reduced sequential combination) can significantly improve the effectiveness and performance of software model checking.

The approach of using information from previous verification runs to guide later testing attempts [6] was instantiated for the sequential combination of the verifier Dafny and the testing tool Pex [11]. This work outlines several promising scenarios where such an approach can potentially improve the precision and preformance of the overall analysis and ensure small test suites for reduced testing effort compared to the traditional testing approach (Pex alone). We are convinced that a future experimental evaluation of this CMC configuration also reveals significant improvements.

Parallel combinations (reduced products) were investigated in the past (e.g., [7, 16, 19]) and can be successfully applied if the overall algorithms of the analyses are similar. Such combinations are complementary to sequential combinations. However, our goal is to investigate the possibilities of combining analysis techniques that are not necessarily similar, available as source code, or require the same platform. Sequential combination with conditions can bring together highly decoupled components, e.g., verification tools that do not need to run at the same time, on the same machine, or at the same location (cloud computing).

Another way of parallel combination (concurrent execution) is to run the same analysis on different parts of the state space in parallel on a multi-core machine [17, 27]. This is orthogonal to conditional model checking, as we make no assumption on how each of the analyses is executed, allowing the use of parallel algorithms. Even distributed model checking (e.g., [2, 30]) can be used as a sub-analysis of a sequential combination. The performance improvement would be independent.

So-called incremental or regression model checking also re-uses analysis results in order to make subsequent runs easier [24, 29, 34, 35], but it uses the results for verifying a different program with the same analysis. It uses the results of successful verification runs by storing (summaries of) the successfully verified part of the state space. On the contrary, conditional model checking verifies the same program with a different analysis after an unsuccessful verification run.

2. CONDITIONAL MODEL CHECKING

In conditional model checking (CMC), a model checker needs to output a condition that summarizes the work that it has performed so far (which parts of the program were successfully verified and which parts were not yet checked). This condition needs to be written in a format that makes it possible to use it as input condition for another model checker. In the following, we define one such format and describe how it can be used for conditional model checking. However, conditional model checking does not require this specific condition format. Any format can be used as long as the involved model checkers support it. For example, the code annotations presented by Christakis et al. [11] might be used in their CMC instance.

The technical details and formal definitions of our analyses are described in a technical report [6] on CMC, and in our previous work on the CPA framework [7–9].

2.1 Output Conditions

We define the following format for the condition, which is easy to obtain in model checkers that are based on abstract reachability trees (ART). We reduce the ART that the model checker produces to an assumption automaton, which is annotated with generated assumptions. Each transition of the assumption automaton corresponds to a control-flow edge and is labeled with the assumption that was used by the model checker when processing this edge. An assumption is produced for a transition if the analysis decided to not explore a certain path of the state space, or if it failed to establish a full safety proof for the part of the program that follows the CFA edge that the transition corresponds to. The assumption $false$ is also added to transitions that lead to frontier states that the analysis did not explore in case of a premature termination (e.g., due to resource exhaustion). Subtrees of the ART for which all transitions are labeled with the assumption $true$ (i.e., the subtrees could be completely verified), are collapsed into a single sink state 'T', which has a self edge that matches any control-flow edge and is labeled with the assumption $true$. For all other parts of the ART there is a 1:1 correspondence between ART nodes and automaton states.

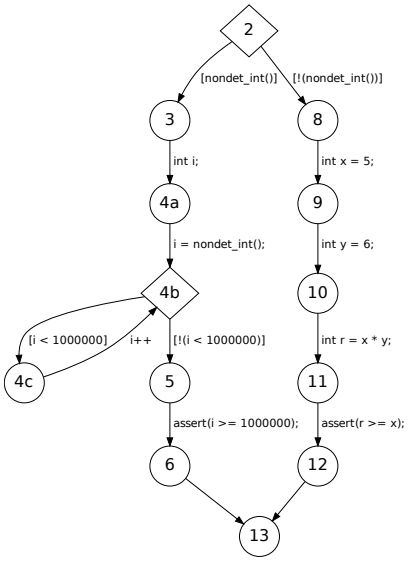

Figure 4: Control-flow automaton for the example program with non-linear safety condition (cf. Fig. 3)

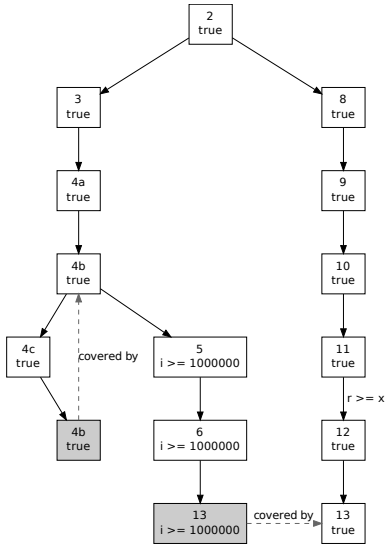

Figure 5: ART for the example program; covered states are gray, edges are labeled with assumptions

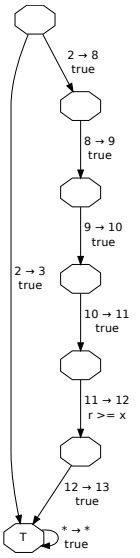

Figure 6: Assumption automaton for the example program

2.2 Conditions as Input

An assumption automaton as defined above can be used as an input condition. Note that, as a condition states which parts of the state space were successfully verified, the second model checker needs to check those states that do not satisfy the condition. For an assumption automaton, this means that the second verifier has to analyze only those paths of the program that contain at least one edge for which the corresponding transition is not labeled with the assumption *true*. We implemented this restriction by running the automaton in parallel to the analysis as an own 'abstract domain' when exploring the control-flow automaton, processing a control-flow edge whenever the automaton can take a matching transition. The exploration of a path can be stopped as soon as the automaton reaches the sink state 'T', because this means that the assumption of all future edges is always *true*.

2.3 Example

We show the conditions produced during the analysis of the example program in Fig. 3. The control-flow automaton can be seen in Fig. 4. This program can neither be verified by predicate analysis (due to imprecision caused by non-linear arithmetic) nor by explicit analysis (due to lack of resources), but it can be successfully verified by a sequential combination of both, if the condition that summarizes the work done by the predicate analysis is used as input condition for the explicit analysis.

Fig. 5 shows the ART that the predicate analysis generates. Each ART node is labeled with the control-flow node that it belongs to and the formula that represents the abstract state. The ART edges are labeled with the assumptions that the analysis produced during the verification run. In this case, there is only one non-trivial assumption at the edge from program location 11 to 12. For all other edges, the assumption is *true* and not shown in the ART. The assumption automaton that the analysis outputs for this example is shown in Fig. 6. Each transition of the automaton is labeled with a control-flow edge (e.g., "2 → 8" is the edge from location 2 to 8) and the assumption that was used when taking this edge. It can be seen that one part of the state space (ART nodes 3 to 6) was collapsed into the sink state 'T' as this whole subtree has no assumption different from *true*.

The second model checker will now let the assumption automaton from Fig. 6 run in parallel to the state-space exploration of the explicit analysis. Whenever the automaton enters the sink state 'T' on a path, the analysis of this path will be stopped. Thus the loop will be skipped and only the second branch of this program will be analyzed. The explicit-value analysis needs to prove only the safety of the second assertion, which it can check efficiently. The final analysis correctly reports that the program is safe.

2.4 Failure-Preventing Conditions

We also implemented two techniques to restrict the search of the model checker via conditions and to prevent failure. First, we monitor the progress for every component of the model checker. When such a component reaches a certain limit (e.g., exhausts its assigned time or memory), the monitor discovers that the limit is reached, terminates the component, and generates a condition that excludes the corresponding states from the verification result (we use such a condition for restricting the time resource in Sect. 3.3). Second, we implemented several conditions that try to predict situations in which the model checker should not continue to try verifying this part of the program, and generate conditions excluding that part (bounded model checking is one such possibility to restrict, or bound, the search). Such conditions make it possible to verify larger parts of the program, instead of spending all time on a particular, unsolvable problem. Thus, if we cannot completely verify a program, we at least obtain a "verification coverage" that is as large as possible, and we can use the resulting condition to continue the verification with different tools and configurations.

3. EXPERIMENTAL RESULTS

Our experiments investigate the practical benefits of conditional model checking when applied to the sequential combination of model checkers with information passing. We show that this approach can improve the overall verification performance (i.e., the total sum of consumed analysis resources) and also the effectiveness (number of solved verification problems).

3.1 Benchmark Programs

The benchmark programs that we used in the experiments are taken from, or constructed from, the benchmark verification problems of the competition on software verification [3]. CPAchecker, the verification platform that we used for implementing conditional model checking, participated in the competition, and there are two categories where it has significant potential for improvement: the categories 'DeviceDrivers64' and 'SystemC'. Many programs in these categories could not be verified due to either a timeout or memory-out. Therefore, we concentrate on benchmark programs from these two 'hard' categories. The first set of benchmark programs in our experiments (cf. Table 1) contains instances from the category 'SystemC'.

In order to create larger and even more difficult programs, we combined programs from category 'DeviceDrivers64' with the program `mem_slave_tlm.1` from category 'SystemC' (a program that cannot even be verified by CPAchecker's predicate analysis). The combination of programs was done by concatenating the program sources and creating the following new main function that calls the main functions of the original programs:

```
1 void main() {
2   // use uninitialized variable
3   // for non-deterministic value
4   int nondet;
5
6   if ( nondet) main1();
7   if (!nondet) main2();
8 }
```

The name of each new program indicates both the original programs and the order in which they are called. The results for this second benchmark set are given in Table 2.

3.2 Experimental Setup and Reporting

All experiments were performed on a machine with a 3.4 GHz Quad Core CPU and 16 GB of RAM. The operating system was Ubuntu 10.04 (64 bit), using Linux 2.6.35 as kernel and OpenJDK 1.6 as Java virtual machine. We used CPAchecker from revision 5871 of the repository for running the experiments. A time limit of 15 minutes and a memory limit of 15 GB were used. CPAchecker was configured with a Java heap size of 12 GB. The predicate analysis uses MathSAT 4.2.17 as SMT solver.

All benchmark programs, the used configurations, scripts and a ready-to-run version of CPAchecker are available on the supplementary web page `http://www.sosy-lab.org/~dbeyer/cpa-cmc`.

In all configurations, CPAchecker was instructed to check every error report using CBMC 4.1 (i.e., whenever CPAchecker finds an error path, it generates a C program for the path and queries CBMC about the feasibility of it). In cases where CBMC determines a path as infeasible, the analysis continues in order to check for the existence of a different feasible error path; if later no feasible error path is found, the analysis result is 'unknown' instead of 'program is safe' (because there might be some abstract states that are covered by an infeasible error path that was not analyzed). The time necessary for running CBMC is included in the reported verification times. This counterexample check with CBMC is done in order to increase the precision (and to avoid reporting false positives). For example, the explicit-value analysis of CPAchecker cannot store inequalities (e.g., facts of the form '$x \neq 0$') or relations between variables. Such facts are necessary to verify some of the benchmark programs. The predicate analysis is limited by the chosen theories for the predicate abstraction (linear arithmetics and equality with uninterpreted function symbols).

For each run, the tables show the consumed processor time of the verifier (in seconds and rounded to two significant digits) and a symbol for the result. The symbol '✓' indicates that the model checker computed the correct answer to the verification problem (bug found or safe). A question mark means that the model-checking configuration was too imprecise to verify the program (model checker returned 'unknown'). This occurred only with the explicit analysis. The symbols T and M indicate a timeout and an out-of-memory condition, respectively. There was no experiment for which the verifier computed a wrong answer (no false-positives/false alarms, no false-negatives/missed bugs). Programs whose name contains BUG are known to be unsafe. The column 'LOC' shows the number of lines of each program. In the last row of a table, we report the total runtime of each configuration, and how many verification problems were successfully solved.

Figure 7 gives a performance overview for all successful results using a plot of the quantile functions. The function graph for a configuration yields the maximum run time y for the x fastest computed correct results. For example, the plot shows that the 40th fastest result of the configuration 'Expl. (100 s)+Pred.' needed about 100 s. This means that this configuration could successfully verify 40 programs in under 100 s each, and took longer than that for all remaining programs. The x-value for which a graph ends at the top gives the maximal number of successfully verified programs for the configuration. The area below a graph (its integral) represents the accumulated verification time that the configuration needed for all programs that it could verify.

3.3 Configurations

We experimented with several verification configurations of the tool CPAchecker, which is an open verification platform with support for explicit-value analysis and predicate analysis. We configure CPAchecker to perform an explicit-value analysis (column 'Explicit'), which keeps track of integer values explicitly while searching in depth-first order through the state space. In this configuration, the abstract states at meet points of the control flow are not joined, in order to obtain a precise analysis. However, such a precise analysis usually exhausts the given memory quickly for large programs, and may get stuck in loops since no abstractions or summaries are computed. We also configure CPAchecker for a predicate analysis with lazy abstraction [25], CEGAR [13], and adjustable-block encoding (ABE) [9] (column 'Predicate'). This is a powerful but expensive analysis, consuming large amounts of resources when verifying large programs. The predicate analysis is based on SMT solving and Craig interpolation for linear arithmetics.

For comparison, we also report results that a simple sequential combination of explicit-value and predicate analysis would yield, where the explicit-value analysis is started first, and then the predicate analysis is performed, without passing information from the explicit-value analysis to the predicate analysis (column 'Expl.+Pred.'). If the first analysis terminates with a final answer ('bug found' or 'program is safe'), the verification is finished. Otherwise, a predicate analysis can be started and consume the remaining time, and the final result is given by the last configuration. The total verification time is the sum of the run times of both configurations.

To make this combination more interesting, we give to the explicit-value analysis as input the condition that it should terminate after 100 s of time (column 'Expl. (100 s)+Pred.'). If the analysis could not complete the verification task in the given time, the predicate analysis is started. This is expected to waste less time on examples for which the explicit-value analysis would fail to verify even with more time, and instead give more time to the predicate analysis.

The most interesting configuration is a sequential combination of explicit-value and predicate analysis with passing information from the first analysis to the second, leveraging the features of conditional model checking (column 'CMC'). For this, we run the explicit-value analysis with an input condition that specifies a time limit of 100 s. After this time, the analysis will terminate gracefully, dumping its verification results as a condition that describes the verified parts of the state space of the program. Then the (conditional) predicate analysis is started, which reads the condition of the explicit-value analysis and accordingly verifies the remaining parts. The condition is created as an assumption automaton (cf. Fig. 6) that summarizes those parts of the abstract reachability tree that were verified. All subtrees that were completely verified are omitted.

The second analysis reads the condition file and runs the automaton in parallel, as explained in Sect. 2. Whenever the model checker analyzes an edge of the control-flow automaton, the assumption automaton takes the matching condition. When the assumption automaton indicates that an edge does not need to be verified, the analysis can stop exploring that path. As additional information, the tables report the size of the condition that is passed from the explicit-value to the predicate analysis, by printing the number of states of the automaton in the last column.

3.4 Discussion

The explicit-value analysis of CPAchecker is able to verify 38 programs with a total runtime of 27 000 s. It successfully verifies only 10 programs that are safe (half of which are only variants of a single program). However, it is able to find the bug in many programs that are declared to contain a bug. In most cases in which this configuration produces a result, it is quite fast (only a few seconds of verification time). The amount of time used for all successfully verified 38 programs is only 2 300 s. This is the only configuration that terminates on some programs without producing a result and without exceeding the available resources. This occurs if it has analyzed the whole state space but found only counterexamples that CBMC proved infeasible. As such infeasible counterexamples may have masked real counterexamples, the analysis cannot give an answer in this case.

The predicate analysis is more successful: it verifies a total of 58 programs using 31 000 s of verification time. However, there are 14 programs that the explicit-value analysis could verify, but the predicate analysis can not. This shows that it is a good idea to combine several analyses.

The results of the simple sequential combination of different analyses (col. 'Expl.+Pred.') show that a simple combination without passing information is not the right way. This naive combination is able to verify 51 programs, less than the predicate analysis alone, and still uses more time (34 000 s). This is because the explicit analysis wastes too much time on many programs, and the predicate analysis thus has no chance to work on them sufficiently.

The other simple combination (col. 'Expl. (100 s)+Pred.'), which limits the explicit-value analysis to at most 100 s, is already better than any of the previous combinations. The number of verified programs is 67 and the run time is 24 000 s. This was expected, because the different techniques have different strengths and weaknesses. For example, explicit-value analysis is considered better in detecting shallow bugs and predicate analysis might be better in proving the absence of safety violations.

Now we leverage the strengths of conditional model checking for an even better sequential combination of different approaches: we use conditions to transfer information from one analysis to the next. By having the first (conditional) model checker inform the second (conditional) model checker about what has already been verified, the second model checker has to check a much smaller state space. The second verifier can concentrate on the partial problems that the first verifier could not solve, whereas in previous combinations, all work had to be done again.

The experiments demonstrate (col. 'CMC') that the application of conditional model checking —implementing the two analyses in such a way that they produce verification results as condition and accept conditions as input— can significantly improve the performance as well as the number of solved instances. Compared to the best configuration so far (col. 'Expl. (100 s)+Pred.'), this combination is able to verify 8 more programs in only 56% of the time.

The total number of verified programs is 75 and the total verification time is only 14 000 s. Several verification problems that cannot be verified by any other configuration are now verified in under 200 s, for example the combinations of `mem_slave_tlm.1` with `loop.BUG`, `usbcore.BUG`, and `kbtab`. There are seven programs that cannot be handled by this analysis, although they can be verified by the 'Expl. (100 s)+Pred.' configuration. In these cases, the input condition lets the predicate analysis actually do more work than it would without. This is because CPAchecker's adjustable-block encoding usually merges many abstract states, which is prevented when it uses a large assumption automaton that is produced by the explicit-analysis (which does never merge its abstract states). However, this is not a principal disadvantage of conditional model checking, but only due to the specific format used for the condition. More research work can, and should, be done in this direction, i.e., to improve the effectiveness of the conditions.

The last column in the tables shows the number of states of the assumption automaton that is created by the explicit-value analysis. This may give an indication of how large the remaining parts of the state space are. For example, if this number is 1, then the explicit analysis proved the program

Table 1: Results for verification tasks from the category 'SystemC'
(times given in seconds, condition size in number of states of the assumption automaton;
✓: correct answer, ?: result 'unknown', T: timeout, M: out-of-memory)

		Explicit		Predicate		Expl.+Pred.		Expl. (100 s) +Pred.		CMC		
Program	LOC	Time		Time		Time		Time		Time		Cond. size
kundu1_BUG	511	1.7	✓	6.1	✓	1.7	✓	1.7	✓	2.2	✓	598
kundu2_BUG	615	1.4	✓	6.3	✓	1.4	✓	1.4	✓	1.6	✓	498
pc_sfifo_1_BUG	359	> 900	T	1.5	✓	> 900	T	100	✓	120	✓	333265
pc_sfifo_2_BUG	464	> 900	T	1.6	✓	> 900	T	100	✓	120	✓	395121
pipeline_BUG	813	12	?	15	✓	27	✓	27	✓	98	✓	72742
token_ring.01.BUG	481	1.5	?	2.6	✓	4.1	✓	4.1	✓	2.8	✓	405
token_ring.02.BUG	606	1.8	?	4.6	✓	6.4	✓	6.4	✓	3.3	✓	793
token_ring.03.BUG	731	2.3	?	6.7	✓	9.0	✓	9.0	✓	4.8	✓	1668
token_ring.04.BUG	856	2.8	?	17	✓	20	✓	20	✓	6.1	✓	3661
token_ring.05.BUG	981	3.5	?	89	✓	93	✓	93	✓	9.6	✓	8178
token_ring.06.BUG	1106	4.7	?	100	✓	110	✓	110	✓	13	✓	18319
token_ring.07.BUG	1231	8.0	?	> 900	T	> 900	T	> 900	T	21	✓	40860
token_ring.08.BUG	1356	19	?	> 900	T	> 900	T	> 900	T	46	✓	90505
token_ring.09.BUG	1481	61	?	> 900	T	> 900	T	> 900	T	120	✓	198966
token_ring.14.BUG	1851	490	✓	> 900	T	490	✓	> 900	T	> 900	T	846
toy1.BUG	711	1.5	✓	92	✓	1.5	✓	1.5	✓	1.7	✓	539
toy2.BUG	699	1.5	✓	89	✓	1.5	✓	1.5	✓	1.7	✓	536
transmitter.01.BUG	444	1.4	✓	2.1	✓	1.4	✓	1.4	✓	1.5	✓	237
transmitter.02.BUG	568	1.6	✓	2.5	✓	1.6	✓	1.6	✓	1.9	✓	373
transmitter.03.BUG	692	1.6	✓	4.3	✓	1.6	✓	1.6	✓	2.1	✓	537
transmitter.04.BUG	816	2.1	✓	8.4	✓	2.1	✓	2.1	✓	2.2	✓	729
transmitter.05.BUG	940	2.1	✓	8.7	✓	2.1	✓	2.1	✓	2.7	✓	949
transmitter.06.BUG	1064	2.5	✓	25	✓	2.5	✓	2.5	✓	2.7	✓	1197
transmitter.07.BUG	1188	3.2	✓	80	✓	3.2	✓	3.2	✓	3.7	✓	1473
transmitter.08.BUG	1312	5.2	✓	520	✓	5.2	✓	5.2	✓	5.7	✓	1777
transmitter.09.BUG	1436	9.4	✓	> 900	T	9.4	✓	9.4	✓	11	✓	2109
transmitter.10.BUG	1560	29	✓	> 900	T	29	✓	29	✓	32	✓	2469
transmitter.11.BUG	1684	110	✓	> 900	T	110	✓	> 900	T	110	✓	2857
transmitter.12.BUG	1808	480	✓	> 900	T	480	✓	> 900	T	> 900	T	874
transmitter.15.BUG	1932	490	✓	3.2	✓	490	✓	100	✓	130	✓	916
transmitter.16.BUG	2057	> 900	T	3.3	✓	> 900	T	100	✓	130	✓	963
bist_cell	499	1.3	✓	2.1	✓	1.3	✓	1.3	✓	1.3	✓	1
kundu	630	6.0	✓	14	✓	6.0	✓	6.0	✓	6.7	✓	1
mem_slave_tlm.1	1363	1.8	✓	> 900	T	1.8	✓	1.8	✓	2.0	✓	1
mem_slave_tlm.2	1368	1.7	✓	> 900	T	1.7	✓	1.7	✓	1.9	✓	1
mem_slave_tlm.3	1373	1.8	✓	> 900	T	1.8	✓	1.8	✓	2.4	✓	1
mem_slave_tlm.4	1378	2.0	✓	> 900	T	2.0	✓	2.0	✓	2.2	✓	1
mem_slave_tlm.5	1382	2.0	✓	> 900	T	2.0	✓	2.0	✓	2.6	✓	1
pc_sfifo_1	359	> 900	T	4.4	✓	> 900	T	100	✓	310	✓	334009
pc_sfifo_2	464	> 900	T	4.3	✓	> 900	T	100	✓	230	✓	389195
pc_sfifo_3	566	1.6	✓	2.0	✓	1.6	✓	1.6	✓	1.7	✓	1
pipeline	813	12	?	88	✓	100	✓	100	✓	630	M	72742
token_ring.01	469	1.4	?	4.3	✓	5.7	✓	5.7	✓	2.5	✓	392
token_ring.02	594	1.6	?	7.3	✓	8.9	✓	8.9	✓	4.0	✓	779
token_ring.03	719	2.0	?	49	✓	51	✓	51	✓	6.9	✓	1652
token_ring.04	844	2.5	?	320	✓	320	✓	320	✓	9.4	✓	3641
token_ring.05	969	3.3	?	> 900	T	> 900	T	> 900	T	19	✓	8150
token_ring.06	1094	4.5	?	750	M	760	M	760	M	43	✓	18275
token_ring.07	1219	7.2	?	> 900	T	> 900	T	> 900	T	150	✓	40784
token_ring.08	1344	19	?	> 900	T	> 900	T	> 900	T	580	✓	90365
toy	703	> 900	T	200	✓	> 900	T	300	✓	> 900	T	397349
Total time / **#Solved**		7200	**26**	17000	**34**	14000	**38**	11000	**41**	5700	**47**	

Table 2: Results for verification tasks from the category 'DeviceDrivers64' combined with 'mem_slave_tlm.1.c'
(times given in seconds, condition size in number of states of the assumption automaton;
✓: correct answer, ?: result 'unknown', T: timeout, M: out-of-memory)

		Explicit		Predicate		Expl.+Pred.		Expl. (100s) +Pred.		CMC		
Program	LOC	Time		Time		Time		Time		Time		Cond. size
mem_slave+drbd.BUG	84078	> 900	T	590	✓	> 900	T	690	✓	240	✓	3746
mem_slave+farsync.BUG	13311	> 900	T	620	✓	> 900	T	720	✓	370	✓	135473
mem_slave+gigaset.BUG	29123	> 900	T	190	✓	> 900	T	290	✓	160	✓	25427
mem_slave+iowarrior.BUG	8549	> 900	T	390	✓	> 900	T	490	✓	170	✓	184755
mem_slave+keyspan_remote.BUG	7279	4.2	✓	> 900	T	4.2	✓	4.2	✓	4.5	✓	2115
mem_slave+lirc_imon.BUG	7934	> 900	T	> 900	T	> 900	T	> 900	T	190	✓	38668
mem_slave+loop.BUG	11244	> 900	T	> 900	T	> 900	T	> 900	T	170	✓	60314
mem_slave+mISDN_core.BUG	30135	6.3	✓	> 900	T	6.3	✓	6.3	✓	8.1	✓	2554
mem_slave+pktcdvd.BUG	15368	> 900	T	450	✓	> 900	T	550	✓	320	✓	39782
mem_slave+ppp_generic.BUG	16828	> 900	T	> 900	T	> 900	T	> 900	T	170	✓	119668
mem_slave+synclink_gt.BUG	24177	6.7	?	140	✓	150	✓	150	✓	53	✓	2634
mem_slave+ttusb_dec.BUG	14356	6.3	✓	420	✓	6.3	✓	6.3	✓	5.9	✓	3125
mem_slave+usbcore.BUG	56379	> 900	T	> 900	T	> 900	T	> 900	T	160	✓	9447
mem_slave+usbmouse.BUG	6252	100	✓	280	✓	100	✓	380	✓	100	✓	2540
mem_slave+i915	130290	220	✓	> 900	T	220	✓	> 900	T	> 900	T	81706
mem_slave+kbtab	6034	> 900	T	> 900	T	> 900	T	> 900	T	170	✓	13372
mem_slave+pppox	6776	> 900	T	> 900	T	> 900	T	> 900	T	190	✓	225378
drbd.BUG+mem_slave	84078	87	✓	90	✓	87	✓	87	✓	87	✓	2504
gigaset.BUG+mem_slave	29123	> 900	T	8.9	✓	> 900	T	110	✓	120	✓	25276
iowarrior.BUG+mem_slave	8549	> 900	T	6.0	✓	> 900	T	110	✓	140	✓	177650
keyspan_remote.BUG+mem_slave	7279	4.1	✓	88	✓	4.1	✓	4.1	✓	4.7	✓	1447
lirc_imon.BUG+mem_slave	7934	> 900	T	40	✓	> 900	T	140	✓	150	✓	38000
loop.BUG+mem_slave	11244	> 900	T	16	✓	> 900	T	120	✓	420	M	57256
mISDN_core.BUG+mem_slave	30135	7.3	✓	79	✓	7.3	✓	7.3	✓	8.7	✓	1886
pktcdvd.BUG+mem_slave	15368	> 900	T	140	✓	> 900	T	240	✓	> 900	T	36549
ppp_generic.BUG+mem_slave	16828	> 900	T	17	✓	> 900	T	120	✓	130	✓	117639
synclink_gt.BUG+mem_slave	24177	8.1	✓	9.1	✓	8.1	✓	8.1	✓	9.3	✓	1830
ttusb_dec.BUG+mem_slave	14356	5.8	✓	6.8	✓	5.8	✓	5.8	✓	5.7	✓	2457
usbcore.BUG+mem_slave	56379	> 900	T	10	✓	> 900	T	110	✓	120	✓	8653
usbmouse.BUG+mem_slave	6252	100	✓	110	✓	100	✓	210	✓	100	✓	1872
btmrvl+mem_slave	11195	> 900	T	190	✓	> 900	T	290	✓	> 900	T	640271
i915+mem_slave	130290	140	✓	> 900	T	140	✓	> 900	T	120	✓	1
kbtab+mem_slave	6034	> 900	T	380	✓	> 900	T	480	✓	> 900	T	12821
pppox+mem_slave	6776	> 900	T	450	✓	> 900	T	550	✓	> 900	T	209219
Total time / **#Solved**		20000	**12**	14000	**24**	20000	**13**	13000	**26**	8400	**28**	

safe. For programs where the first analysis already finds a bug, the automaton consists mostly of the found counterexample and is typically small (up to 3 000 states). If the automaton is large (more than 100 000 states), then conditional model checking is sometimes slower than the simple sequential combination due to the overhead caused by the automaton. The reason for this is mostly that our predicate analysis would usually merge abstract states, but the explicit analysis does not because it would be too imprecise otherwise. With the input condition being read in from the automaton, the predicate analysis is now forced to build an ART similar to that of the explicit analysis, and does not merge states (at least for those states that are present in the automaton). This shows that the power of conditional model checking can be further increased in the future by investigating different formats for the output and input conditions. However, even now there are programs for which a large automaton is indeed useful, for example, the program `mem_slave+pppox`, where the automaton has more than 200 000 states and CMC is the only successful configuration.

Figure 7 helps to interpret the overall results. First, the left part of the plot shows the strengths of the explicit-value analysis: many programs can be verified within a few seconds. In contrast, predicate analysis alone needs more time even for programs that are 'easy' for it. The disadvantage of the explicit analysis is that the verification time increases rapidly as shown by the graph, and more than half of the programs cannot be verified. The advantage of combining analyses with different strengths increases with the complexity of the verification task, i.e., the more difficult the problem is to solve, the better it is to use combination analyses — the most difficult programs can only be solved by combinations. As expected, the two configurations 'Expl.+Pred.'

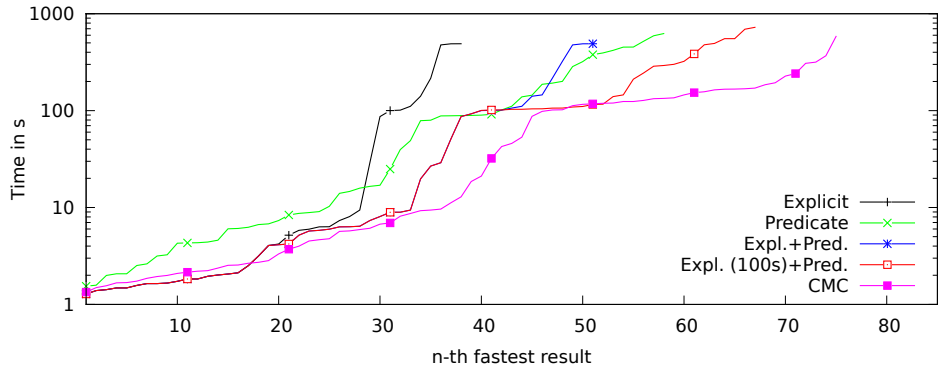

Figure 7: Quantile functions. For each configuration, we plot all pairs (x, y) such that the maximum run time of the x fastest results is y. The graphs are plotted on a logarithmic scale and decorated with symbols at every tenth data point in order to make the graphs distinguishable on gray-scale prints.

and 'Expl. (100 s)+Pred.' differ only for verification times of more than 100 s. The latter of these two configurations is already better in terms of time and successful verification results than predicate analysis alone. It combines the strengths of both configurations, providing fast results for easy programs and being able to solve a large amount of programs within the time and memory limits.

The best graph in the plot refers to the sequential combination with passing of information, based on conditional model checking. On the left (where the explicit analysis is able to solve the programs alone) we notice a small overhead, but the additional work pays off by successfully verifying more programs in less time. There are only few programs for which this configuration needs large amounts of time: 70 programs out of a total of 75 verified programs are analyzed in less than 200 s each.

4. CONCLUSION

Software model checking is an undecidable problem, and therefore, we cannot create model checkers that always give a precise answer to the verification problem. Conventional model checkers fail when they cannot give a precise answer, leaving the user no information about what the tool was not able to verify, where in the program the problem occurred, and how much of the program was already verified. Conditional model checking proposes to design model checkers that do not fail, but instead summarize their work when they decide to give up. That is, we change the outcome from {safe, unsafe, fail} to {condition}, meaning that the model checker has verified that the program satisfies the specification under the reported condition. In addition to this user feedback, conditional model checking improves *verification coverage* and *performance* by making it possible to give conditions as input that avoid certain parts of the program.

We investigated one major application of conditional model checking: the combination of different verifiers with passing information between the verification runs. After the first verification run, the resulting condition is given as input to the second verification run, which uses a different algorithm or abstract domain, and therefore can be expected to fail on different parts of the state space. This process can be repeated until the desired coverage is achieved. In contrast to other combination techniques, e.g., parallel combination by reduced products, the sequential combination with information passing is possible even if the techniques require different iteration algorithms, are implemented in different tools, are only available as binaries, compile only on incompatible platforms, or run at different locations (e.g., in cloud computing).

The experiments confirmed the following benefits of sequential combination with conditional model checking:

1. More problem instances can be solved.
2. The performance can be improved, in terms of reduced run-time and reduced memory consumption.
3. Better coverage of the state space can be achieved for problem instances for which neither the presence nor the absence of an error can be proved.
4. A powerful and flexible combination technique for different verification technologies is established.

Other applications of conditional model checking include partial verification, automatic generation of verifiable benchmarks from programs that are too hard for current techniques, regression model checking, the qualitative comparison of model checking results of different tools, and guided test-case generation based on verification results of static-analysis runs. We also plan to investigate further applications of conditional model checking. For example, conditional model checking can support the verification of components and modules in isolation, and then be used to compose the global verification goal of the system from the partial results (i.e., modeling assumptions and guarantees as 'conditions'). We leave it for future work to further investigate the structure of conditions and its influence on the verification performance, and to provide an even more flexible language for condition exchange between different model checkers.

5. REFERENCES

[1] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.

[2] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Proc. FMCAD*, LNCS 1954, pages 427–441. Springer, 2000.

[3] D. Beyer. Competition on software verification (SV-COMP). In *Proc. TACAS*, LNCS 7214, pages 504–524. Springer, 2012.

[4] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. ICSE*, pages 326–335. IEEE, 2004.

[5] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.

[6] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional Model Checking. Technical Report MIP-1107, University of Passau, 2011.

[7] D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*, pages 29–38. IEEE, 2008.

[8] D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.

[9] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.

[10] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS*, LNCS 1579, pages 193–207. Springer, 1999.

[11] M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *Proc. FM*, 2012, to appear.

[12] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proc. Logic of Programs 1981*, LNCS 131, pages 52–71. Springer, 1982.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[14] E. M. Clarke, D. Kröning, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, LNCS 2988, pages 168–176. Springer, 2004.

[15] C. L. Conway, D. Dams, K. S. Namjoshi, and C. Barrett. Pointer analysis, conditional soundness, and proving the absence of errors. In *Proc. SAS*, pages 62–77. Springer, 2008.

[16] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In *Proc. ASIAN'06*, LNCS 4435, pages 272–300. Springer, 2008.

[17] M. B. Dwyer, S. G. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proc. ICSE*, pages 3–12. IEEE, 2007.

[18] M. B. Dwyer, J. Hatcliff, R. Robby, C. S. Pasareanu, and W. Visser. Formal software analysis emerging trends in software model checking. In *Proc. FOSE*, pages 120–136. IEEE, 2007.

[19] J. Fischer, R. Jhala, and R. Majumdar. Joining data flow with predicates. In *Proc. ESEC/FSE*, pages 227–236. ACM, 2005.

[20] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245. ACM, 2002.

[21] A. Groce and W. Visser. Heuristics for model checking Java programs. *Int. J. Softw. Tools Technol. Transfer*, 6(4):260–276, 2004.

[22] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: A new algorithm for property checking. In *Proc. FSE*, pages 117–127. ACM, 2006.

[23] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proc. CAV*, LNCS 2404, pages 526–538. Springer, 2002.

[24] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. Extreme model checking. In *International Symposium on Verification: Theory and Practice*, LNCS 2772, pages 332–358. Springer, 2003.

[25] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.

[26] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. CAV*, LNCS 1427, pages 440–451. Springer, 1998.

[27] G. J. Holzmann, R. Joshi, and A. Groce. Tackling large verification problems with the SWARM tool. In *Proc. SPIN*, LNCS 5156, pages 134–143. Springer, 2008.

[28] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[29] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *Proc. ICSE*, pages 291–300. IEEE, 2008.

[30] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. SPIN*, LNCS 1680, pages 22–39. Springer, 1999.

[31] M. Musuvathi and D. R. Engler. Model checking large network-protocol implementations. In *Proc. Networked Systems Design and Implementation*, pages 155–168. USENIX, 2004.

[32] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proc. ISSTA*, pages 229–239. ACM, 2002.

[33] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. Symposium on Programming*, LNCS 137, pages 337–351. Springer, 1982.

[34] O. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *Proc. CAV*, LNCS 818, pages 351–363. Springer, 1994.

[35] G. Yang, M. B. Dwyer, and G. Rothermel. Regression model checking. In *Proc. ICSM*, pages 115–124. IEEE, 2009.