Reducer-Based Construction of Conditional Verifiers

Dirk Beyer
LMU Munich, Germany

Marie-Christine Jakobs*
LMU Munich, Germany

Thomas Lemberger
LMU Munich, Germany

Heike Wehrheim*
Paderborn University, Germany

ABSTRACT

Despite recent advances, software verification remains challenging. To solve hard verification tasks, we need to leverage not just one but several different verifiers employing different technologies. To this end, we need to exchange information between verifiers. Conditional model checking was proposed as a solution to exactly this problem: The idea is to let the first verifier output a *condition* which describes the state space that it successfully verified and to instruct the second verifier to verify the yet unverified state space using this condition. However, most verifiers do not understand conditions as input.

In this paper, we propose the usage of an off-the-shelf construction of a *conditional verifier* from a given traditional verifier and a reducer. The reducer takes as input the program to be verified and the condition, and outputs a residual program whose paths cover the unverified state space described by the condition. As a proof of concept, we designed and implemented one particular reducer and composed three conditional model checkers from the three best verifiers at SV-COMP 2017. We defined a set of claims and experimentally evaluated their validity. All experimental data and results are available for replication.

CCS CONCEPTS

• **Software and its engineering → Formal methods**; **Formal software verification**;

KEYWORDS

Conditional Model Checking, Formal Verification, Testing, Program Analysis, Software Verification, Sequential Combination

ACM Reference Format:
Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, and Heike Wehrheim. 2018. Reducer-Based Construction of Conditional Verifiers. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180259

*This author was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

ICSE 2018, May 27 – June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180259

1 INTRODUCTION

Software model checking [47] has received lots of attention in academia and industry [2, 48] in the past two decades — yet, there are many programs that are in principle verifiable, but no existing verifier can solve them automatically. There are many different approaches, but none is superior. The competition on software verification (SV-COMP) [5] gives a yearly overview over the state of the art, in terms of both strengths of verifiers on various categories and weaknesses as shown by a large amount of unsolved problems.

One promising idea is to combine the strengths of different verifiers by condition passing, which was formalized as *conditional model checking* (CMC) [10] six years ago. The idea is simple and effective: The first verifier reports what it had successfully verified and summarizes its work done as a *condition*. The next verifier reads the condition and verifies only the part of the state space not yet covered by the condition. This technique was shown to be effective, and sometimes even more efficient. Unfortunately, it is difficult to write a verifier that can parse the complicated conditions and effectively reduce the state space of the verifier. This complication is responsible for the situation that the technique is not as widely applied as it could be: only a few conditional model checkers exist.

To solve this problem, we developed an automatic construction template that can be used to construct a conditional verifier from a given arbitrary classical verifier. The original work proposed to run a product analysis that guides the state-space exploration such that it concentrates on the state space not covered by the condition. We propose an alternative solution, inspired by earlier work on conditional model checking and testing [35]: We define a *program reducer*, which takes as input a program and a condition, and computes a program whose executions are restricted to those not yet covered by the given condition. Having developed this component once, it is easy to construct a new conditional verifier using the equation CMC = $V \circ R$, where R is the reducer, V is an arbitrary verifier, and $\circ$ is the sequential application of first R to a given program and then V to the output program of R. The new verifier CMC is a conditional verifier that takes as input a program and a condition. Figure 1 illustrates this composition visually. There can be different implementations of reducers, and the reducers might leverage a notion of abstraction, causing the residual program to be more compact but less precise. We implemented a reducer that is based on a product construction, i.e., program and condition

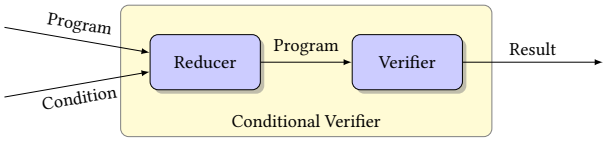
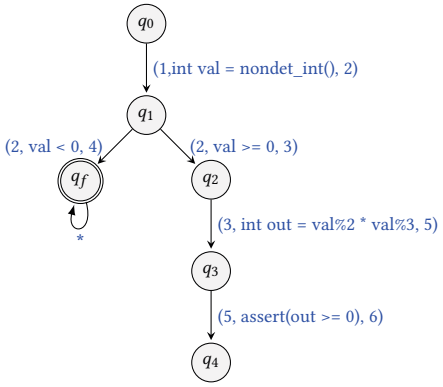


Figure 1: Construction of a conditional verifier

```
1  int val = nondet_int();
2  if (val >= 0) {
3    int out = val%2 * val%3;
   }
   else {
4    int out = -val;
   }
5  assert (out >= 0);
6
```




```
1  int val = nondet_int();
2  if (val >= 0) {
3    int out = val%2 * val%3;
4    assert (out >=0);
5  }
6  else
7    ;
```

(a) Source code **(b) Condition** **(c) Residual program**

Figure 2: Example: (a) fragment of a C program, (b) condition generated by CPAchecker with accepting states as double circles, assumptions elided (all *true*), label * subsuming all control-flow edges, and (c) residual program constructed by Reducer

are converted into automata and the reduced product automaton is converted back to a program.

In our study, we show that the construction works and is effective. We do not claim that our implementation of the reducer is the best possible, but we show for a number of verifiers how to increase the number of obtained results with the reducer-based construction of conditional verifiers. The approach can in some cases even reduce the resource consumption.

Contributions. We make the following contributions:

- We provide a reducer that understands more extensive conditions than a reducer that was previously used in the context of conditional model checking and testing [35].
- We construct a number of conditional verifiers from existing verifiers in order to experimentally show that new combinations with condition passing can significantly increase the number of verified programs.
- We apply the concept also to test-case generation and show that the construction effectively works.
- Our reducer and all experimental data are available for other researchers and practitioners for replication or to strengthen their own verification infrastructure by using newly constructed conditional verifiers that were not available before.

2 CONDITION-BASED REDUCERS

The objective of our work is the construction of conditional verifiers. Conditional verifiers are verification tools accepting programs together with conditions as input. A conditional verifier should check the parts of the program not covered by the condition. To this end, we employ *reducers* constructing residual programs from conditions. We start with giving a formal account of conditions and reducers. In our notation, we follow previous work [11].

2.1 Foundations

Programs are represented by *control-flow automata*[1] (CFAs) $C = (L, \ell_0, G)$ that consist of a set of locations L, an initial location ℓ_0, and a set of control-flow edges $G \subseteq L \times Ops \times L$, where Ops is the set of operations. Intuitively, a program and its CFA are semantically equivalent because the CFA contains exactly the operations of the program on its control-flow edges and in exactly the same order. Our construction of reducers relies on soundly converting programs to CFAs and back within tools. We let $\mathcal{C}$ be the set of all CFAs. In our presentation, we consider operations from a simple programming language, with assume operations and assignments on integer variables. Our implementation covers C programs.

We let X be the set of variables occurring in the operations Ops. A concrete data state c is thus a mapping of X to $\mathbb{Z}$. A *concrete program path* of a CFA $C = (L, \ell_0, G)$ is a sequence $(c_0, \ell_0) \xrightarrow{g_1} \dots \xrightarrow{g_n} (c_n, \ell_n)$ such that c_0 assigns 0 to all variables, $g_i = (\ell_{i-1}, op_i, \ell_i) \in G$, and $c_{i-1} \xrightarrow{op_i} c_i$, i.e., (a) in case of assume operations, $c_{i-1} \models op_i$ (op_i is the assumption) and $c_{i-1} = c_i$, and (b) in case of assignments, $c_i = \mathrm{SP}_{op_i}(c_{i-1})$, where SP is the strongest-post operator of the operational semantics. From a concrete program path $\pi = (c_0, \ell_0) \xrightarrow{g_1} \dots \xrightarrow{g_n} (c_n, \ell_n)$, we can derive an *execution* $ex(\pi) = c_0 c_1 \dots c_n$. We let $path(C)$ be the set of all concrete program paths and $ex(C)$ be the set of executions of a CFA C. A CFA C is *deterministic* (and hence representable as a C program) if the following holds for all $\ell \in L$, $(\ell, op_1, \ell_1), (\ell, op_2, \ell_2) \in G$: either $op_1 = op_2$ and $\ell_1 = \ell_2$, or op_1 is an assume operation and $op_1 \wedge op_2$ is unsatisfiable.

Conditions subsume the results of verification runs on programs. A condition basically states which paths have been explored. In addition, a condition might involve *assumptions* under which the verifier has explored a certain path. Assumptions are given as state conditions (from a set Φ). We write $c \models \varphi$ to say that a concrete state c satisfies a state condition φ.

Definition 2.1. A *condition automaton* (CA) (short: condition) $A = (Q, \Sigma, \delta, q_0, F)$ consists of

- a finite set Q of *states* and an initial state $q_0 \in Q$,
- an *alphabet* $\Sigma \subseteq 2^G \times \Phi$,
- a *transition relation* $\delta \subseteq Q \times \Sigma \times Q$, and
- a set $F \subseteq Q$ of *accepting states*,

and satisfies the following well-formedness condition:

$$\neg\exists (q_f, *, q) \in \delta \text{ with } q_f \in F \wedge q \notin F\ .$$

[1] CFAs are a variant of control-flow graphs [1], with operations attached to the edges.

We let $\mathcal{A}$ be the set of condition automata. Accepting states in conditions are used to describe paths of the CFA which have already been successfully verified. Figure 2 shows an example C program and a condition automaton as generated by CPAchecker. The condition shows that the verifier explored the else-branch of the if-statement (path leading to accepting state q_f) and successfully verified the assertion to hold on that path. Due to the non-linear arithmetic, the verifier could not handle the then-branch, which hence appears in the automaton on a path not entering q_f.

Definition 2.2. A condition automaton $A = (Q, \Sigma, \delta, q_0, F)$ *covers* a path $\pi = (c_0, \ell_0) \xrightarrow{g_1} (c_1, \ell_1) \xrightarrow{g_2} \dots \xrightarrow{g_n} (c_n, \ell_n)$ if there is a run $\rho = q_0 \xrightarrow{(G_1,\varphi_1)} q_1 \xrightarrow{(G_2,\varphi_2)} \dots \xrightarrow{(G_k,\varphi_k)} q_k, 0 \le k \le n$, in A, s.t.

(1) $q_k \in F$,
(2) $\forall i, 1 \le i \le k : g_i \in G_i$, and
(3) $\forall i, 1 \le i \le k : c_i \models \varphi_i$.

The task of a reducer is now the generation of a new program that contains the paths of the original program except for (at most) those already covered by the condition.

Definition 2.3. A *reducer* is a mapping $red : \mathcal{C} \times \mathcal{A} \to \mathcal{C}$ satisfying the following *residual condition*:

Res. $\forall C \in \mathcal{C}, \forall A \in \mathcal{A}$:
$ex(C) \setminus \{ex(\pi) \mid A \text{ covers } \pi\} \subseteq ex(red(C, A)) \subseteq ex(C)$.

In the following, we refer to the output of a reducer as the *residual program*. Note that the Identity relation on CFAs, i.e., $red(C, A) = C$, is a reducer, though not a very effective one. Note also that – contrary to Czech et al. [35] – the residual condition **Res** is not specific to safety properties, i.e., unreachability of error locations. It simply states a coverage property for the residual program. Our definitions allow us to use conditions and reducers as a means for various combinations of verifiers. As one example, both the condition generating verifier **A** as well the condition processing verifier **B** could be tools generating test vectors, and together they manage to achieve complete code coverage. Tools **A** and **B** could, on the other hand, also both be formal software verifiers proving validity of assertions, and together they prove safety of the program.

2.2 Implementation

A reducer takes as input a program (in the form of a CFA) together with a condition automaton and returns a residual program. Note that the definition of reducers gives us some freedom in constructing residual programs, in particular, there is more than one residual program possible. Here, we will present one such reducer.

Our reducer builds upon the idea of Czech et al. [35]. It constructs the residual program by means of a parallel composition of original program and condition, cutting off paths whenever the condition has reached an accepting state. The construction called Reducer is given in Alg. 1. In contrast to Czech et al. [35], Alg. 1 employs an additional residual state q_r to subsume states that the condition automaton either has not investigated, or has investigated but under a non-*true* assumption. Note that Czech et al. do not need q_r because they consider a restricted class of conditions, which, e.g., only considers *true* assumptions. Depending on the condition, the reduction might restructure the program as to isolate paths which need to be cut off. In our example (Fig. 2), the generated

Algorithm 1 Reducer

Input: CFA $C = (L, \ell_0, G)$ ▷ *original program*
CA $A = (Q, \Sigma, \delta, q_0, F)$ s.t. $q_r \notin Q$ ▷ *condition automaton*
Output: CFA $C_r = (L_r, \ell_{0,r}, G_r)$ ▷ *residual program*

1: $L_r := \{(\ell_0, q_0)\}$; $\ell_{0,r} := (\ell_0, q_0)$; $G_r := \emptyset$;
2: waitlist := L_r;
3: **while** waitlist $\neq \emptyset$ **do**
4: choose $(\ell_1, q_1) \in$ waitlist; remove (ℓ_1, q_1) from waitlist;
5: **for each** $g = (\ell_1, op, \ell_2) \in G$ **do**
6: **if** $q_1 \in Q \wedge \exists (q_1, (G_1, true), q_2) \in \delta$ s.t. $g \in G_1$ **then**
7: **for each** $(q_1, (G_1, true), q_2) \in \delta$ s.t. $g \in G_1$ **do**
8: **if** $q_2 \notin F \wedge (\ell_2, q_2) \notin L_r$ **then**
9: waitlist := waitlist $\cup \{(\ell_2, q_2)\}$;
10: $L_r := L_r \cup \{(\ell_2, q_2)\}$;
11: $G_r := G_r \cup \left\{\left((\ell_1, q_1), op, (\ell_2, q_2)\right)\right\}$;
12: **else**
13: **if** $(\ell_2, q_r) \notin L_r$ **then**
14: waitlist := waitlist $\cup \{(\ell_2, q_r)\}$;
15: $L_r := L_r \cup \{(\ell_2, q_r)\}$;
16: $G_r := G_r \cup \left\{\left((\ell_1, q_1), op, (\ell_2, q_r)\right)\right\}$;
17: **return** C_r

condition describes that paths taking the else-branch have been successfully verified while paths taking the then-branch still need to be explored. Hence, the reducer generates a residual program where the assertion is moved inside the then-branch so as to ensure that the assertion need not be checked again for the else-branch.

Theorem 2.4. *Algorithm* Reducer *is a reducer.*

Proof. Assume C, A, C_r as used in Alg. 1. We have to show $ex(C) \setminus \{ex(\pi) \mid A \text{ covers } \pi\} \subseteq ex(C_r) \subseteq ex(C)$.
We separately look at the two set inclusions:

$ex(C_r) \subseteq ex(C)$: Let $c_0 \dots c_n \in ex(C_r)$. Then, there exists a path $\pi = (c_0, (\ell_0, q_0)) \xrightarrow{g_1} \dots \xrightarrow{g_n} (c_n, (\ell_n, q_n)) \in path(C_r)$ such that $g_i = ((\ell_{i-1}, q_{i-1}), op_i, (\ell_i, q_i))$ and $c_{i-1} \xrightarrow{op_i} c_i$. From this, we inductively construct a path π' of C (and hence the execution of C):

- Induction start: take $\pi' = (c_0, \ell_0)$.
- Induction step: assume path π' to be constructed up to some $(c_j, \ell_j), j < n$.
  We know that $g_{j+1} = ((\ell_j, q_j), op_{j+1}, (\ell_{j+1}, q_{j+1})) \in G_r$ (as π is a path of C_r). New elements are inserted into G_r in lines 11 and 16 of the algorithm only, while iterating over elements of G (line 5). Hence $(\ell_j, op_{j+1}, \ell_{j+1}) \in G$, and we can extend π' by $(c_j, \ell_j) \xrightarrow{(\ell_j, op_{j+1}, \ell_{j+1})} (c_{j+1}, \ell_{j+1})$.

$ex(C) \setminus \{ex(\pi) \mid A \text{ covers } \pi\} \subseteq ex(C_r)$: Let $c_0 \dots c_n \in ex(C) \setminus \{ex(\pi) \mid A \text{ covers } \pi\}$. Then, there is a path $\pi = (c_0, \ell_0) \xrightarrow{g_1} \dots \xrightarrow{g_n} (c_n, \ell_n)$ of C that is not covered by A. Note that thus $q_0 \notin F$, as otherwise all paths are covered. We inductively construct a path $\pi' = (c_0, (\ell_0, q_0)) \xrightarrow{g'_1} \dots \xrightarrow{g'_n} (c_n, (\ell_n, q_n))$ of C_r with $g'_i = ((\ell_{i-1}, q_{i-1}), op_i, (\ell_i, q_i))$ together with a run $\rho = q_0 \xrightarrow{(G_1,\varphi_1)} \dots \xrightarrow{(G_m,\varphi_m)} q_m$ of A s.t. $0 \le m \le n$.

They satisfy the following properties: (a) $\forall i, 0 \leq i \leq n : q_i \notin F$, (b) $\forall i, 0 \leq i < n$: (ℓ_i, q_i) is an element of `waitlist` at some point in time during the algorithm, and (c) at position m the path is split into two parts, the second of which may be empty, such that: (i) $\forall i \leq m : q_i \neq q_r, i = 0 \vee \varphi_i = true \wedge g_i \in G_i$, and (ii) $\forall j > m : q_j = q_r$.

- Induction start: take $\pi' = (c_0, (\ell_0, q_0))$ and $\rho = q_0$. Then, $q_0 \notin F$, $q_0 \neq q_r$, and (ℓ_0, q_0) is initially in `waitlist`.
- Induction step: assume path π' to be constructed up to some $(c_j, (\ell_j, q_j)), j < n$, and ρ up to some $q_l, l < m$.

  We know that $g_{j+1} = (\ell_j, op_{j+1}, \ell_{j+1}) \in G$ and $c_j \xrightarrow{op_{j+1}} c_{j+1}$ (as π is path of C). We have two cases to consider:

  (1) $q_j \neq q_r$: Hence, $q_j \in Q$ and by induction hypothesis, $q_j \notin F$, $q_j = q_l, j = l$. Again two cases:
  - (a) $\exists (q_j, (G_{j+1}, true), q_{j+1}) \in \delta, g_{j+1} \in G_{j+1}$ (line 6): We extend π' by $(c_j, (\ell_j, q_j)) \xrightarrow{g_{j+1}} (c_{j+1}, (\ell_{j+1}, q_{j+1}))$ and ρ by $(q_j, (G_{j+1}, true), q_{j+1})$. We have $q_{j+1} \notin F$ as the path π is not covered by A and ρ would witness coverage otherwise. Hence, (ℓ_{j+1}, q_{j+1}) is added to `waitlist` (unless it has been in there before). We stay in the first part of the path.
  - (b) Else (line 12): We switch to the second part of the path. We extend the path π' by $(c_j, (\ell_j, q_j)) \xrightarrow{g_{j+1}} (c_{j+1}, (\ell_{j+1}, q_r))$ and let ρ remain unchanged. We have $q_r \notin F$ (as it is an extra state) and (ℓ_{j+1}, q_r) is added to `waitlist` (unless it has been in there before).

  (2) $q_j = q_r$: Then, we are in the second part of the path and proceed as in (1), case (b). □

To be usable by the condition processing verifier, the residual CFA has to be transformed back into a C program. The residual CFA obtained by Reducer from a deterministic CFA (i.e., a C program) is again deterministic since the condition generated by CPAchecker is always deterministic. Moreover, note that we currently inline procedure calls. Thus, Reducer may fail on recursive programs.

3 REDUCER-BASED VERIFIERS

In the previous section, we introduced two reducers, Identity and Reducer. Next, we introduce the second component of our conditional verifiers, the off-the-shelf tools that we transform into conditional verifiers. In this paper, we transform four verifiers and three test-generation tools. As verifiers, we use the best three tools CPAseq, Smack, and Ultimate Automizer from SV-COMP 2017 [5] (Table 1 gives an overview). Additionally, we use the value analysis from the CPAchecker framework [15], which supports condition automata as input conditions (an in-tool CMC solution [10]) and allows us to compare the concept of reducer-based conditional verifiers against an in-tool solution. As test-generation tools, we chose AFL-fuzz, Crest-ppc, and Klee. All three are open source and have lately attracted high interest by research [17, 23, 27, 49, 55, 56, 59]. In the next paragraphs, we explain the technologies underlying the selected verifiers and test-generation tools.

Value Analysis. CPAchecker's value analysis is a configurable program analysis [11]. Its reachability analysis tracks the values of certain variables of interest explicitly while assuming that the remaining variables may have any possible value. The precision [11] is increased iteratively, based on counterexample-guided abstraction refinement (CEGAR) [31] and lazy refinement [43]. To get the best refinement, the analysis applies refinement selection [20]. Given an infeasible error path, path-prefix slicing [21] is used to compute different overapproximations of the error path s.t. each overapproximation replaces some assume operations with no-ops. For each overapproximation, interpolation [18] is used to compute a refinement candidate. In the end, the best refinement is selected.

CPAseq. CPAseq uses the CPAchecker framework [15] to run four different analyses in sequence. Whenever an analysis gives up (due to timeout or unknown result), the next analysis starts. A definite answer (feasible error path or proof) of an analysis is returned immediately. CPAseq starts with a simple value analysis without refinement, which tracks all variable values immediately. Next, a value analysis similar to the one described above is used. The third analysis is a bit-precise predicate analysis [16] that uses adjustable-block encoding [16] to compute predicate abstractions only at loop heads. The set of predicates is determined by a combination of interpolation [42] and CEGAR [31] with lazy refinement [43]. The last analysis runs k-induction in parallel with invariant generation [9]. The invariants found so far are used to improve the k-induction step and are provided by numerical and predicate analyses.

Smack. The Smack [54] verifier consists of a translation front end and a verification back end. First, it translates the input program to Boogie code (via intermediate LLVM code). Based on heuristics, the Boogie code is either verified with Boogie or Corral. Boogie [3] proves a verification condition generated with the weakest precondition calculus. Corral [50] tries to find a property violation with a two-staged CEGAR approach. First, it uses variable abstraction to compute an overapproximation of the program, which only considers a subset of the program variables. The variable abstraction is adapted whenever the second CEGAR approach fails to rule out an infeasible error path. On the second stage, Corral inlines functions (summaries) up to a given recursion depth (loops are assumed to be written as recursive functions). Functions are only inlined if the function summary appears in an infeasible error path.

Ultimate Automizer. Ultimate Automizer (UAutomizer) [40, 41] uses an automata-based verification approach. In principle, it maintains an overapproximation of error paths in form of an automaton. A CEGAR approach successively refines the overapproximation, i.e., it removes infeasible error paths, until a feasible error path is found or the automaton language is empty. In each refinement step, a generalization of an infeasible error path is excluded from the current overapproximation. The generalization of the error path is described by a Floyd-Hoare automaton [41], which associates boolean formulas over predicates with its states. The initial state is associated with *true*, accepting states are associated with *false*, and transitions describe valid Hoare triples. The predicates used in the Hoare triples are obtained via interpolation along the infeasible error path.

AFL-fuzz. AFL-fuzz is a random fuzzing tester. Given a set of start inputs, it performs different mutations on the existing inputs, executes these newly created inputs, and checks whether new program parts are explored. If this is the case, the inputs are kept and used for further mutation. Otherwise, the inputs are discarded. [2]

[2] AFL (American Fuzzy Lop) is available at http://lcamtuf.coredump.cx/afl/.

Table 1: Overview of applied verification technologies in the verifiers

Verifier	Technique	Refinement			Bitprecise
		CEGAR	Lazy abstraction	Interpolation	
CPAseq	ARG, explicit and numerical values, predicates, k-induction	✓	✓	✓	✓
Smack	property-driven reachability [24], bounded model checking [22]	✓	✓	×	✓
UAutomizer	automata, predicates	✓	✓	✓	✓

Klee. Klee [26] uses symbolic execution for test-case generation. Symbolic execution is an extension to concrete execution of a program. For every unknown input value to a program, a new symbolic value is introduced that initially represents any possible value. During execution, the symbolic values are constrained by branching conditions along the program (e.g., if-branches in a C program). These constraints are used to compute whether a given program path is feasible, and which class of input values will lead to executions that take this path. Whenever both branches are feasible in a symbolic execution, Klee copies its current symbolic execution state and continues to explore one branch with the current state and the other with the copied state. After each step in a program, Klee heuristically chooses with which of the existing execution states to continue. Given several heuristics, Klee alternates between them.

Crest-ppc. Crest-ppc [49] is an improved version of Crest [25]. Crest uses concolic execution for testing and provides different heuristics to achieve higher code coverage. Concolic execution is a combination of symbolic execution and concrete execution. A program under test is executed with concrete inputs that determine one concrete execution path. In parallel, a symbolic execution is performed on that path to obtain constraints over program inputs on this path. Based on these *path constraints*, a constraint solver computes new inputs that lead to the execution of another, yet unvisited program part. New executions are performed and new inputs are generated until all program parts are explored. Crest uses heuristics to choose which unvisited program part to explore next. To increase the performance, Crest-ppc adds a heuristic to Crest that submits more calls to the constraint solver but uses fewer constraints per call.

4 EVALUATION

4.1 Claims to be Evaluated

In the following, we list our claims and how we plan to evaluate them. The claims are not on efficiency, but on effectiveness. That is, we provide means for solving additional verification tasks by investing more computing resources, but without implementing or changing verification tools.

Feasibility Hypothesis. A reducer can be used to effectively construct conditional verifiers from existing verification tools. *Evaluation Plan:* We show this by implementing one particular instance of a reducer, and apply our reducer-based construction of conditional verifiers to three model checkers and three testers. The result is a set of six conditional verifiers, and we take standard configurations "out of the box", without changing a single line of the verifiers.

Null Hypothesis. Applying a reducer has no effect. *Evaluation Plan:* We compare the results using our reducer against the results using the identity function as replacement for the reducer.

Claim 1. Reducer-based conditional verification is not much worse then "native" conditional verification. *Evaluation Plan:* The original proposal of CMC [10] implements the restriction of the state space that the condition describes internally in the exploration engine of the verifier. We claim that it also works reasonably well to use an external reducer instead, which opens the door for constructing new conditional verifiers without actual implementation work.

Claim 2. The technique of conditional verification can effectively increase the number of overall solved verification tasks if additional resources are provided. *Evaluation Plan:* We select a number of hard-to-solve verification tasks and perform experiments on them using the original verifiers and the constructed verifiers.

Claim 3. Conditional verification with condition passing can solve verification tasks that neither CPAseq, Smack, nor Ultimate Automizer can solve. *Evaluation Plan:* We select from a given set of verification tasks those verification tasks that none of the original verifiers, but at least one of the conditional verifiers can solve.

Claim 4. The use of different conditional verifiers improves the overall effectiveness. *Evaluation Plan:* We report results for different conditional verifiers and consider verification tasks that only one conditional combination can solve.

Claim 5. Reducer-based conditional verification is also applicable to test-case generation. *Evaluation Plan:* We construct conditional verifiers from three test-generation tools and compare the number of generated crashing tests against the result of the test-generation tools alone.

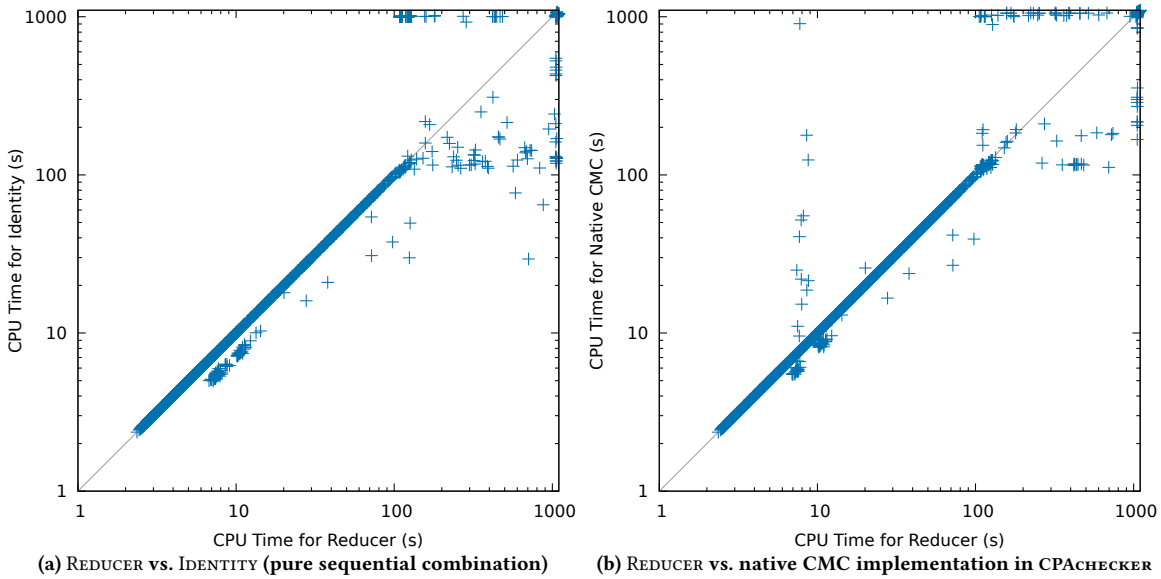
4.2 Setup

Computing Resources. We performed our experiments on machines with an Intel Xeon E3-1230 v5 CPU with 8 processing units each, a frequency of 3.4 GHz, 33 GB of memory, and an Ubuntu 16.04 operating system with Linux kernel 4.4. We limited each analysis run to 15 GB of memory and a varying time limit, depending on the experiment, and allowed it to use all 8 processing units. We report CPU time and memory use with two significant digits.

Verification Tasks. To get a representative set of verification tasks, we used all 5 687 programs from ReachSafety categories of the SV-COMP benchmark set[3] in revision cc49668 [4]. For all input programs, we verify the property that function __VERIFIER_error is never called. A total of 1 501 of the 5 687 programs are unsafe, i.e., the call to __VERIFIER_error is reachable, and 4 186 programs are safe.

Tools. We used a predicate analysis for condition generation and a value analysis for comparison with native conditional model checking, both from the CPAchecker project. Our implementation of a reducer is also available in the CPAchecker project. For all experiments, we used CPAchecker from branch reducer-patch in revision r25656. For the verifiers in the composition of our reducer with a verifier, we use the three best tools from SV-COMP 2017, as submitted to the competition[5] (without any modifications) and

[3] https://sv-comp.sosy-lab.org/2017/benchmarks.php
[4] https://github.com/sosy-lab/sv-benchmarks
[5] https://sv-comp.sosy-lab.org/2017/systems.php



(a) Reducer **vs.** Identity **(pure sequential combination)** **(b)** Reducer **vs. native CMC implementation in** CPAchecker

Figure 3: Comparison of CPU time of different CMC solutions for predicate (100s) + value analysis

the three test-generation tools described previously. To streamline the testing process for the test-generation tools, we use the testing framework tbf [17] [6] in revision b60a924. We run our experiments with BenchExec [19] (version 1.14).[7]

Availability. All our experimental data are available online [14].[8]

4.3 Experiments

Feasibility Hypothesis. We designed and implemented a proof-of concept reducer, and licensed the reducer using the open-source license Apache 2.0 such that other researchers can later use it. While our implementation certainly has potential for improvement, we show that the approach of composing a conditional verifier from an arbitrary verifier and our reducer works in practice. We demonstrated this by using the three best verifiers directly from the SV-COMP web site and composed the conditional verifiers without any change to the verifiers. In addition, we also composed conditional verifiers from test-generation tools, in order to help test-generators to produce crashing tests for more verification tasks.

Null Hypothesis. We have experimented with verification runs in which we replaced our reducer by an identity function Identity, i.e., the reducer is effectively removed from the tool chain. The first verifier, which generates the condition, is a predicate analysis that we restrict to at most 100 s of CPU time. For the second verifier, we use CPAchecker's value analysis with a time limit of 900 s.

Figure 3a uses a scatter plot to illustrate the CPU times of the reducer-based approach using Reducer (x-axis) against using Identity (i.e., pure sequential combination). The scatter plot shows results only for those verification tasks that at least one of the two combinations can solve and that none of them solved incorrectly or crashed on. Thus, the plot only displays results that have a useful result. Often, the results are similar (data points close to the diagonal). In this case, the predicate analysis alone already solved the verification task. For some tasks, Reducer is slower or even times out, due to the large size of residual programs. The reason is that Reducer restructures the program, e.g., unfolds loops and the program structure. The residual program becomes much larger and more complex in its structure, which complicates the task of the second verifier in these cases. However, there are also a set of tasks for which Reducer is significantly faster: the data points close to the upper border represent tasks for which the conditional combination with Reducer solved the task while the combination with Identity timed out. Thus, the null hypothesis is rejected.

Claim 1 (Comparison against native implementation). We compare our proposed reducer-based approach to construct conditional verifiers against the approach of the original implementation [10], which we refer to as 'native' approach because it implements the restriction of the state space according to the condition internally in the verifier. We use the same setup as above, but replace the second verifier by CPAchecker's value analysis with the internal condition treatment enabled. Figure 3b shows the useful results as scatter plot, again. Most of the data points are close to the diagonal, i.e., the two solutions perform similarly. However, as above, when the residual program gets too large, the reducer-based solution sometimes uses too much time (right side). For some tasks, the reducer-based solution is even faster than the native approach (top). Thus, Claim 1 is valid.

Claim 2 (Effective increase of number of verified programs). We now evaluate the claim that the use of two complementing verifiers joined by reducer-based conditional verification can effectively solve additional verification problems if additional resources are spent on running a combination after the runs of the original verifier. In our experiments, we always first run a conditional verifier based on predicate analysis to output the conditions, with a time limit of 100 s of CPU time. The predicate analysis combines lazy abstraction refinement [43] with predicate abstraction with adjustable-block encoding (ABE) [16]. ABE is configured to abstract

[6] https://github.com/sosy-lab/tbf
[7] https://github.com/sosy-lab/benchexec
[8] https://www.sosy-lab.org/research/reducer/

Table 2: Results of using a verifier on its own vs. a combination with predicate analysis and condition passing

	CPAseq	Smack	UAuto	Predicate + CPAseq	Predicate + Smack	Predicate + UAuto
Correct	513	415	238	789	695	789
Correct proof	265	76	170	387	296	386
Correct alarm	248	339	68	402	399	403
Incorrect	0	0	7	0	0	4
Incorrect proof	0	0	4	0	0	0
Incorrect alarm	0	0	3	0	0	4
Unknown	307	405	575	31	125	27
Total	820	820	820	820	820	820

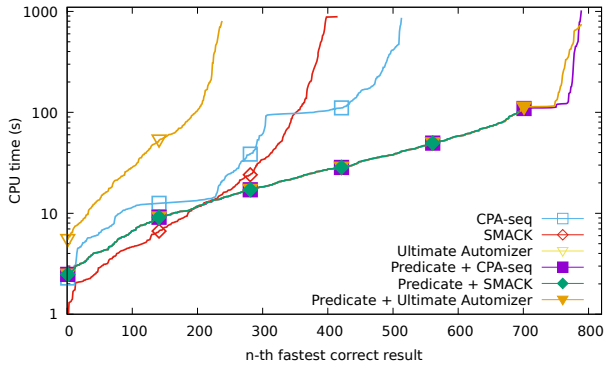


Figure 4: Quantile plots for the six verification approaches

at loop heads only. Let us refer to this verifier as **A**. The conditional verifier in the second verification step is always constructed from our reducer and an off-the-shelf verifier; we limit the CPU time to 900 s. Let us refer to this kind of verifier as **B**. Verifier **B** tries to solve all tasks that **A** was not yet able to solve, with the help of the conditions generated by **A** for these tasks. The time limit of 900 s of CPU time is considered a community standard (cf. SV-COMP), because most verification tasks can either be solved way below this time limit or cannot be solved at all. As verifier **B**, we compose our reducer with the three best tools from SV-COMP 2017 on reachability properties: CPAseq, Smack, and Ultimate Automizer.

Many of the verification tasks in the considered task set from the SV-COMP benchmarks are easy to solve for the standard verifiers. For those tasks, we do not need to further experiment because our aim is to show that the new approach can increase the overall number of verified programs. Therefore, we restrict our experiment to verification tasks that are hard-to-solve; in particular, we select those verification tasks for which at least one verifier **B** fails but the corresponding combination with condition passing of **A** and **B** solves the task. This results in a benchmark set containing 820 hard-to-solve verification tasks.

Table 2 breaks down the effectiveness of each verification approach. It lists the number of verification tasks that each verification approach solved correctly, solved incorrectly, and which it cannot solve ('Unknown'). The correct and incorrect results are further classified into answers that reported a proof and a bug, respectively. Inspecting the numbers, we observe the following: In all three cases, the reducer-based CMC combination with condition passing of verifiers **A** and **B** solves significantly more tasks correctly than verifier **B** alone. At the same time, the number of wrong answers is not increased by the conditional verifier. There are two possible reasons for this improvement: First, verifier **A** already accomplished the verification task, in which verifier **B** has no work (suggested by the data points on the diagonal with less than 100 s in Fig 3a). Or second, verifier **A** verified a significant portion of the verification task such that the residual program generated by Reducer becomes easier to analyze for verifier **B** (suggested by the middle and lower part of Table 3).

Figure 4 shows quantile plots for all six verification approaches. A data point (x, y) on such a graph means that the x fastest correct results can be solved all in max. y s of CPU time each. We observe that all reducer-based approaches significantly outperform their standalone counterpart by investing max. 100 s of CPU time. These observations together with Table 2 validate our Claim 2.

Claim 3 (Solving problems that none of the three can solve). We consider a particular subset of the verification tasks, namely those that none of the verifiers CPAseq, Smack, and Ultimate Automizer can solve as standard verifier but at least one combination can. These tasks seem to be particularly hard for verifiers while not being too hard for our approach. Table 3 shows an excerpt of those 143 programs of the task set. For each verification task (identified by name and expected verification result), the table contains groups of result, CPU time, and max. memory usage, for each of the three standard verifiers and their reducer-based combination with condition passing. From the table, it can be observed that Claim 3 is valid: there exist programs that conditional combinations can solve but none of the given standard verifiers can.

Claim 4 (Different back ends have different strengths). None of the conditional verifiers is superior. Each verifier has its strengths: for two verifiers **B** there exist verification tasks that only a combination with that verifier can solve and no other combination (cf. Table 2). And each verifier has its weaknesses: for each verifier, there are some verification tasks that the verifier, even in combination, cannot solve. To solve all difficult tasks, we need to leverage different technologies. The experimental results validate Claim 4.

This last observation makes the contribution of our reducer-based approach important: It does not make sense to extend existing verifiers to become conditional verifiers (in terms of accepting conditions as inputs), because we need *many* conditional verifiers. Our approach to take an *arbitrary* verifier off-the-shelf and construct a conditional verifier without implementation work significantly improves the overall achieved verification power.

Claim 5 (Reducer-based construction works also for testing). To demonstrate that our approach can be applied to tools other than model checkers, we combine our Reducer with three test-generation tools, namely AFL-fuzz (v2.46b), Crest (revision 31c32f4), and Klee (v1.4.0). As in the other experiments, the first analysis (which generates the condition) is the predicate analysis, again limited to 100 s. The test-generation is limited to 900 s.

Analogous to Claim 2, we restrict the experiment to those verification tasks that are hard-to-solve with test generation: we select those tasks for which at least one test generator fails to uncover a bug in, but that the corresponding combination with condition passing can correctly solve. In addition, since testing cannot prove correctness, we only consider verification tasks that are unsafe.

Table 3: Results of verification tasks for which all considered verifiers A alone could not compute a result, but for which at least one verifier B succeeded in a reducer-based combination with condition passing. Column *R* shows the expected result of the corresponding task: either no property violation exists (T) in the program or a property violation exists (F). Column *S* reports whether the task was solved by the corresponding verifier, *t* is the CPU time in seconds spent to achieve the corresponding result, and *M* the used memory in GB.

Task	R	CPAseq			Smack			UAutomizer			+CPAseq			+Smack			+UAutomizer		
		S	t (s)	M (GB)	S	t (s)	M (GB)	S	t (s)	M (GB)	S	t (s)	M (GB)	S	t (s)	M (GB)	S	t (s)	M (GB)
loop-acc overflow	T	✗	910	7.8	✗	880	0.93	✗	900	1.3	✓	2.6	0.27	✓	2.6	0.27	✓	2.6	0.27
mutex_unbounded	F	✗	910	4.9	✗	0.13	0.021	✗	900	0.94	✓	5.6	0.32	✓	5.6	0.32	✓	5.6	0.32
mutex_unlock	F	✗	320	4.9	✗	0.11	0.020	✗	900	1.2	✓	11	0.47	✓	11	0.47	✓	11	0.47
lin-4.0 legousbtower	F	✗	180	15	✗	880	0.56	✗	900	4.0	✓	12	0.48	✓	12	0.48	✓	12	0.48
lin-4.0 net2272	F	✗	56	15	✗	890	1.0	✗	900	7.3	✓	15	0.53	✓	15	0.53	✓	15	0.53
fib_longer	F	✗	900	3.7	✗	880	0.15	✗	8.6	0.30	✓	15	0.61	✓	15	0.61	✓	15	0.61
lin-3.4 vivi	F	✗	230	15	✗	880	0.25	✗	48	1.3	✓	15	0.59	✓	15	0.59	✓	15	0.59
lin-3.0 block-loop	F	✗	900	8.5	✗	880	0.48	✗	900	3.8	✓	16	0.51	✓	16	0.51	✓	16	0.51
lin-4.2 lm78	T	✗	950	6.8	✗	890	1.2	✗	910	13	✓	18	0.63	✓	18	0.63	✓	18	0.63
lin-3.4 synaptics	F	✗	210	15	✗	880	0.43	✗	900	1.6	✓	18	0.62	✓	18	0.62	✓	18	0.62
lin-3.16 mISDN	T	✗	910	8.8	✗	950	3.0	✗	900	5.7	✓	26	0.96	✓	26	0.96	✓	26	0.96
lin-4.2 vfio	F	✗	910	8.1	✗	890	0.47	✗	900	5.3	✓	26	0.70	✓	26	0.70	✓	26	0.70
val-0.8 g_printer	F	✗	910	8.4	✗	880	0.73	✗	900	5.5	✓	28	0.87	✓	28	0.87	✓	28	0.87
val-0.6 g_printer	F	✗	910	8.4	✗	880	0.71	✗	900	5.6	✓	28	0.85	✓	28	0.85	✓	28	0.85
...																			
Problem19_label20	T	✗	520	15	✗	880	2.8	✗	900	13	✓	110	0.37	✗	110	0.37	✓	110	0.37
Problem19_label57	T	✗	440	15	✗	880	2.9	✗	900	13	✓	110	0.36	✗	110	0.37	✓	110	0.38
Problem19_label37	T	✗	440	15	✗	880	3.2	✗	900	13	✓	110	0.38	✗	110	0.37	✓	110	0.37
Problem19_label15	T	✗	440	15	✗	880	3.0	✗	900	11	✓	110	0.37	✗	110	0.37	✓	110	0.38
Problem19_label44	T	✗	440	15	✗	880	2.9	✗	900	12	✓	110	0.39	✗	110	0.37	✓	110	0.37
Problem19_label36	T	✗	500	15	✗	880	2.9	✗	900	13	✓	110	0.38	✗	110	0.38	✓	120	0.38
Problem19_label06	T	✗	460	15	✗	880	2.9	✗	910	14	✓	110	0.37	✗	110	0.38	✓	110	0.36
Problem19_label56	T	✗	440	15	✗	880	2.9	✗	910	13	✓	110	0.39	✗	110	0.37	✓	110	0.37
Problem19_label30	T	✗	450	15	✗	880	3.2	✗	910	13	✓	110	0.36	✗	110	0.37	✓	110	0.37
Problem19_label01	T	✗	440	15	✗	880	2.9	✗	900	11	✓	110	0.37	✗	110	0.37	✓	110	0.37
Problem19_label09	T	✗	550	15	✗	880	3.0	✗	900	11	✓	110	0.37	✗	110	0.37	✓	110	0.37
Problem19_label40	T	✗	450	15	✗	880	2.9	✗	900	13	✓	110	0.38	✗	110	0.37	✓	110	0.36
Problem13_label33	T	✗	550	15	✗	880	3.1	✗	900	7.2	✓	110	0.29	✗	110	0.30	✓	110	0.32
Problem19_label05	T	✗	450	15	✗	880	2.9	✗	900	12	✓	110	0.38	✗	110	0.37	✓	110	0.36
...																			
lin-4.2 vlsi_ir	T	✗	910	7.9	✗	890	0.97	✗	900	13	✓	490	10	✗	130	0.67	✗	150	0.77
lin-3.14 vsp1	T	✗	920	6.9	✗	890	0.70	✗	910	14	✗	550	1.5	✗	610	1.5	✓	640	1.5
lin-3.14 vxge	T	✗	930	11	✗	190	14	✗	19	0.51	✗	760	1.4	✗	630	1.5	✓	650	1.5
lin-4.2 w83781d	T	✗	910	6.7	✗	900	3.7	✗	910	14	✗	690	1.5	✗	660	1.4	✓	660	1.5
lin-4.2 zd1211rw	T	✗	930	6.3	✗	890	0.96	✗	140	11	✗	720	1.5	✗	670	1.5	✓	660	1.5
lin-3.14 vmxnet3	T	✗	930	6.9	✗	890	1.2	✗	900	10	✗	540	1.5	✗	640	1.4	✓	670	1.4
lin-3.14 skge	T	✗	950	7.3	✗	940	3.6	✗	410	15	✗	650	1.5	✗	600	1.5	✓	670	1.5
lin-3.16 ath5k	T	✗	950	5.9	✗	950	4.7	✗	900	13	✗	710	1.5	✗	730	1.5	✓	710	1.5
lin-3.14 ipw2200	T	✗	950	7.6	✗	950	6.6	✗	15	0.39	✗	700	1.5	✗	730	1.5	✓	720	1.5
lin-3.14 bttv	T	✗	950	5.8	✗	910	5.0	✗	20	0.51	✗	720	1.5	✗	770	1.4	✓	750	1.5
lin-4.2 cciss	T	✗	920	7.1	✗	330	12	✗	900	4.7	✓	790	10	✗	120	0.77	✗	180	5.3
floodmax.4	T	✗	910	3.0	✗	880	0.53	✗	910	13	✓	900	4.3	✗	110	0.42	✗	1 100	7.9
sep20	T	✗	900	3.2	✗	880	0.10	✗	910	13	✓	1 000	2.6	✗	110	0.27	✗	150	0.99
Sum		0	100 k	1 600	0	110 k	500	0	110 k	1 200	120	28 k	180	42	24 k	130	121	25 k	160
Average			720	11		800	3.5		760	8.1		200	1.2		170	0.89		170	1.1

The full version of this table can be found at https://www.sosy-lab.org/research/reducer/ .

Table 4: Test generation vs. CMC combination

	AFL-FUZZ	CREST	KLEE	Predicate + AFL-FUZZ	Predicate + CREST	Predicate + KLEE
Correct alarm	96	44	277	479	476	477
Incorrect proof	0	0	0	0	0	0
Unknown	384	436	203	1	4	3
Total	480	480	480	480	480	480

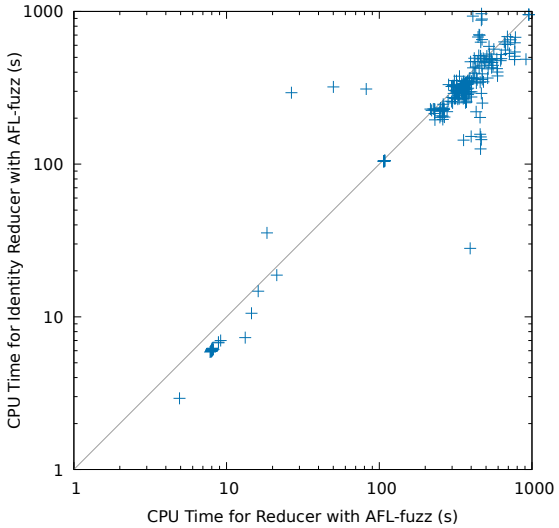


Figure 5: CPU time for predicate analysis and AFL-FUZZ combined with REDUCER (CMC) and with IDENTITY (sequential)

As a result, we get 480 tasks. Table 4 compares the performance of the CMC scenarios with the tester performance. Similar to Table 2, it shows the numbers of correct alarms, incorrect proofs, and unsolved tasks. However, it leaves out the rows related to safe verification tasks. We see that for all three test-generation tools the number of correct alarms of our reducer-based combination with condition passing is higher than for the respective tester. In general, such an improvement is not only caused by the use of the verifier **A**, but often a result of the combination of tools. To further support this statement, we present Fig. 5. It shows the CPU time of two reducer-based CMC solutions, both using the predicate analysis mentioned above to generate conditions, using the full set of 1 501 verification tasks with expected result *false*. The first solution (x-axis) uses the reducer REDUCER with AFL-FUZZ and the second solution (y-axis) uses the IDENTITY reducer with AFL-FUZZ (pure sequential combination). For better visualization, we removed the results that the predicate analysis can solve on its own. Due to the mentioned blowup of the residual program, the REDUCER based solution (REDUCER plus AFL-FUZZ) performs worse for some tasks, but it can also solve a significant amount of tasks faster than the pure sequential combination (IDENTITY plus AFL-FUZZ).

Size of residual programs. As already mentioned, the residual program created by REDUCER may become significantly larger than the original program. The reason is a large amount of branching in the condition, i.e., unfolding of loops and program structure, which is needed to record the verification work already performed. To study this in more detail, we compared the sizes of the original and the residual program in terms of locations in the CFA. At worst, the residual program was more than 10 times larger than the original program (1 934 vs. 22 325 locations). At best, the number of locations in the residual program is less than 1 % of the number of original program locations (200 253 vs. 127 locations). On average, the residual program contains fewer locations (with a mean of 27 % and a median of 14 % of the number of locations in the original program). While the residual program can be much larger, it is often much smaller.

4.4 Threats to Validity

We did not cross-check the reported verification results with an independent verifier because we currently do not know how to construct correctness or violation witnesses [7, 8] in the setting of reducer-based conditional model checking. While we are sure that the standalone verifiers did a proper inspection (they successfully participated in SV-COMP or provide a test), tools might have guessed the correct answer when run as part of the conditional verifier. Yet, we think that guessing is unlikely. The tools are laid out to provide witnesses and thus properly perform their verification.

The correctness of the residual program is another threat. Like other analysis tools, we rely on the soundness of the transformation from program to CFA and back. Additionally, we rely on the soundness of the existing condition generating tool in that the condition only covers paths the verifier has already inspected. Furthermore, our implementation of REDUCER is a prototype which revealed bugs during evaluation. In principle, the bugs might be the reason for the effectiveness of the reducer-based approach. However, the bugs we observed led the conditional verifier to report a wrong result.

Additionally, we checked the null hypothesis and claim 1 only with a single condition generating analysis and a single conditional verifier. Thus, the corresponding results might not be universally valid in any reducer-based conditional-model-checking setup.

When using a combination of two verification tools with CMC, it is also possible that the increase in solvable tasks is simply because the different tools can solve a distinct set of tasks each in a very short time. E.g., in our configuration, it could have been possible that the full increase in additionally solvable tasks is only due to a competence of the predicate analysis in quickly solving a set of tasks that none of the other verifiers can solve. To make sure that our considered tool combinations actually benefit from the use of condition passing, we provided a comparison with a pure sequential combination (IDENTITY) that showed the general benefit. In addition, CPASEQ includes a 200 s run of predicate analysis in its configuration—this ensures that all benefits for our combination of CPASEQ and predicate analysis are actually due to our REDUCER approach. Of the 820 tasks considered in the experiments backing claim 2, 143 cannot be solved by any of the sequential combinations, but only when using CMC with condition passing.

We only consider a subset of the SV-COMP tasks and the three best verifiers from SV-COMP. These three verifiers might be tuned to SV-COMP tasks and may perform worse on our generated residual programs. Despite this possible bias, our approach still improves the existing verifiers. In addition, the three test-generation tools used never participated in any edition of SV-COMP and are unlikely biased. Our approach still shows improvements for them.

5 RELATED WORK

Our concept of reducers allows us to combine a condition generating software verifier with an arbitrary second verifier. Techniques for combining different verification approaches have intensively been studied in the past. The approaches are executed in parallel, interleaved, or sequentially. Orthogonally, the approaches are integrated in a white-box or black-box style. White-box combinations tightly integrate typically orthogonal approaches, whereas black-box combinations aim at a loose coupling of different tools.

Parallel Combinations. Parallel combinations are often used in a white-box style if the analysis algorithms are similar. Typically, combinations [12, 32, 33] let different domains interoperate to obtain analyses that are more precise than a product combination.

Interleaved Combinations. Interleaved combinations are often white-box combinations that unite different techniques in one algorithm. For example, SYNERGY [39] and DASH [4] perform an alternation of test generation and proof construction. Test generation is guided by the abstract error paths and the abstraction for the proof construction is adapted according to the tests. SMASH [38] combines underapproximation with overapproximation. In contrast, abstraction-driven concolic testing [36] is a black-box integration that alternates concolic testing with model checking. The main goal of the model checker is to identify and exclude infeasible paths. Given the open test goals (encoded as error locations), the model checker builds an abstract reachability graph (ARG). The built ARGs successively restrict the (original) program considered by the tester, i.e., after each model-checking run the new program for the tester becomes the intersection of its previous program with the ARG.

Sequential Combinations Testifying Verification Result. Many sequential combinations aim at excluding false alarms after an imprecise static analysis, typically using a black-box combination. For example, Blast [6], Check'n'Crash [34], DyTa [37], and SANTE [28] try to build a test case for each alarm and only report those alarms that are backed by a test. Post et al. [53] and CPAchecker [57] use bounded model checking to check whether an alarm is realizable. Residual investigation [51] tries to reduce the number of false or irrelevant alarms. It only reports alarms for which dynamic analysis observed program behavior indicating that a warning is appropriate. In contrast, proof-carrying code (PCC) approaches [44, 52] check a complete proof. Standardized verifier exchange formats like correctness or error witnesses [7, 8] enable cross-checks between different tools.

Sequential Combinations Splitting Verification Effort. Program partitioning [46] suggests to use the test data to partition the control-flow graph (CFG) into tested and not-tested. The non-tested partition, a subgraph of the CFG, is analyzed by a static analyzer.

Conditional model checking [10] uses a sequential combination: A first verifier constructs a condition summarizing the performed verification, the next verifier uses that condition to steer its verification. We use the same idea for the first verifier, but we transform the condition into a residual program checked by the next verifier.

Multi-goal reachability analysis for testing [13] reuses the verification effort of one (test) goal for another one. The idea is to transform the ARG that was built to achieve the test goal, s.t. it fits for a new test goal. The test-goal automata can be seen as conditions encoding sets of program paths.

Christakis et al. [29, 30] propose that a verifier should add program annotations stating which assertions under which conditions were verified. In the experiments, the static analyzer Clousot produces annotations that guide the exploration of the tester PEX.

Czech et al. [35] use conditions and a residual-program construction to combine model checking and testing in the context of safety checking. They propose two basic program constructions. Their synchronous composition of condition and program is similar to our Reducer. However, they consider a restricted class of conditions and thus do not need to consider assumptions nor program paths that are not covered by the condition. The second approach slices the program for assertions that are not fully verified.

Generating Programs from Verification Results. Program partitioning [46] extracts a subgraph of the program which has not been tested. Abstraction-driven concolic testing [36] computes a program from the intersection of an ARG and a program. A similar idea, namely using ARGs to generate programs, has already been proposed in a PCC context [45, 58]. Czech et al. [35] compute a synchronous combination of condition and program. As already mentioned, our residual-program construction is similar to the approach of Czech et al. [35]. Our implementation constructs an ARG, representing the combination of condition and control-flow graph, which is translated into a program. In contrast to program partitioning [46], the generated programs need not be subgraphs.

6 CONCLUSION

Software verification is an undecidable problem, but still, almost all live-critical systems are controlled by software, and thus, we need to verify these large software systems. One research direction is to develop faster verification algorithms and theories; another direction is to leverage existing results by combinations. Our contribution falls into the second research area. Conditional model checking is a promising approach to combine the strengths of different verifiers. However, it is a large effort to make a verifier understand and use the condition that describes what the first verifier already achieved. To solve this problem, we propose an easy, automatic template construction that turns an off-the-shelf verifier into one that understands conditions. Our idea is to use a preprocessor, the reducer, which takes the condition and the original program to compute a residual program. The residual program encodes the remaining verification task in a format that is understandable by every verifier: program code. In this paper, we suggested one possible reducer. Our experiments on hard tasks of the SV-COMP benchmark collection show that our reducer-based CMC solution is effective. Using the new combination technique, we can solve many verification tasks that were not solvable before, and thus advance the frontier of what is possible with existing software verifiers.

The main conclusion from our experiments is that we need many conditional verifiers, but that it is not worth the effort to change existing verifiers. Rather we can simply apply our construction to get k conditional verifiers from k arbitrary existing verifiers, without changing one line of code. Even if the task is to find crashing test cases with state-of-the-art test-generation tools, we can significantly increase the number of found bugs by using a plug-and-play construction that does not cost any development effort, but increases the number of valuable test cases significantly.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. http://www.worldcat.org/oclc/12285707

[2] T. Ball and S. K. Rajamani. 2002. The Slam Project: Debugging System Software via Static Analysis. In *Proc. POPL*. ACM, 1–3. https://doi.org/10.1145/503272.503274

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proc. FMCO (LNCS 4111)*. Springer, 364–387. https://doi.org/10.1007/11804192_17

[4] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. 2008. Proofs from Tests. In *Proc. ISSTA*. ACM, 3–14. https://doi.org/10.1145/1390630.1390634

[5] D. Beyer. 2017. Software Verification with Validation of Results (Report on SV-COMP 2017). In *Proc. TACAS (LNCS 10206)*. Springer, 331–349. https://doi.org/10.1007/978-3-662-54580-5_20

[6] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. 2004. Generating Tests from Counterexamples. In *Proc. ICSE*. IEEE, 326–335. https://doi.org/10.1109/ICSE.2004.1317455

[7] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. 2016. Correctness Witnesses: Exchanging Verification Results Between Verifiers. In *Proc. FSE*. ACM, 326–337. https://doi.org/10.1145/2950290.2950351

[8] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. 2015. Witness Validation and Stepwise Testification Across Software Verifiers. In *Proc. ESEC/FSE*. ACM, 721–733. https://doi.org/10.1145/2786805.2786867

[9] D. Beyer, M. Dangl, and P. Wendler. 2015. Boosting k-Induction with Continuously-Refined Invariants. In *Proc. CAV (LNCS 9206)*. Springer, 622–640. https://doi.org/10.1007/978-3-319-21690-4_42

[10] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. 2012. Conditional Model Checking: A Technique to Pass Information Between Verifiers. In *Proc. FSE*. ACM, 57. https://doi.org/10.1145/2393596.2393664

[11] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2007. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proc. CAV (LNCS 4590)*. Springer, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

[12] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2008. Program Analysis with Dynamic Precision Adjustment. In *Proc. ASE*. IEEE, 29–38. https://doi.org/10.1109/ASE.2008.13

[13] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. 2013. Information Reuse for Multi-goal Reachability Analyses. In *Proc. ESOP (LNCS 7792)*. Springer, 472–491. https://doi.org/10.1007/978-3-642-37036-6_26

[14] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim. 2018. Replication Package for Article "Reducer-Based Construction of Conditional Verifiers", Proc. ICSE'18. https://doi.org/10.5281/zenodo.1172228

[15] D. Beyer and M. E. Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Proc. CAV (LNCS 6806)*. Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[16] D. Beyer, M. E. Keremoglu, and P. Wendler. 2010. Predicate Abstraction with Adjustable-Block Encoding. In *Proc. FMCAD*. IEEE, 189–197. http://ieeexplore.ieee.org/document/5770949/

[17] D. Beyer and T. Lemberger. 2017. Software Verification: Testing vs. Model Checking. In *Proc. HVC (LNCS 10629)*. Springer, 99–114. https://doi.org/10.1007/978-3-319-70389-3_7

[18] D. Beyer and S. Löwe. 2013. Explicit-State Software Model Checking Based on CEGAR and Interpolation. In *Proc. FASE (LNCS 7793)*. Springer, 146–162. https://doi.org/10.1007/978-3-642-37057-1_11

[19] D. Beyer, S. Löwe, and P. Wendler. 2015. Benchmarking and Resource Measurement. In *Proc. SPIN (LNCS 9232)*. Springer, 160–178. https://doi.org/10.1007/978-3-319-23404-5_12

[20] D. Beyer, S. Löwe, and P. Wendler. 2015. Refinement Selection. In *Proc. SPIN (LNCS 9232)*. Springer, 20–38. https://doi.org/10.1007/978-3-319-23404-5_3

[21] D. Beyer, S. Löwe, and P. Wendler. 2015. Sliced Path Prefixes: An Effective Method to Enable Refinement Selection. In *Proc. FORTE (LNCS 9039)*. Springer, 228–243. https://doi.org/10.1007/978-3-319-19195-9_15

[22] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded Model Checking. *Advances in Computers* 58 (2003), 117–148. https://doi.org/10.1016/S0065-2458(03)58003-2

[23] M. Böhme, V.-T. Pham, and A. Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proc. SIGSAC*. ACM, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[24] A. R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Proc. VMCAI (LNCS 6538)*. Springer, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

[25] J. Burnim and K. Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proc. ASE*. IEEE, 443–446. https://doi.org/10.1109/ASE.2008.69

[26] C. Cadar, D. Dunbar, and D. R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. OSDI*. USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[27] M. Chalupa, M. Vitovská, M. Jonás, J. Slaby, and J. Strejcek. 2017. Symbiotic 4: Beyond Reachability (Competition Contribution). In *Proc. TACAS (LNCS 10206)*. Springer, 385–389. https://doi.org/10.1007/978-3-662-54580-5_28

[28] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. 2012. Program Slicing Enhances a Verification Technique Combining Static and Dynamic Analysis. In *Proc. SAC*. ACM, 1284–1291. https://doi.org/10.1145/2245276.2231980

[29] M. Christakis, P. Müller, and V. Wüstholz. 2012. Collaborative Verification and Testing with Explicit Assumptions. In *Proc. FM (LNCS 7436)*. Springer, 132–146. https://doi.org/10.1007/978-3-642-32759-9_13

[30] M. Christakis, P. Müller, and V. Wüstholz. 2016. Guiding Dynamic Symbolic Execution Toward Unverified Program Executions. In *Proc. ICSE*. ACM, 144–155. https://doi.org/10.1145/2884781.2884843

[31] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2003. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *J. ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643

[32] P. Cousot and R. Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. ACM Press, 269–282. https://doi.org/10.1145/567752.567778

[33] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2006. Combination of Abstractions in the ASTRÉE Static Analyzer. In *Proc. ASIAN (LNCS 4435)*. Springer, 272–300. https://doi.org/10.1007/978-3-540-77505-8_23

[34] C. Csallner and Y. Smaragdakis. 2005. Check 'N' Crash: Combining Static Checking and Testing. In *Proc. ICSE*. ACM, 422–431. https://doi.org/10.1145/1062455.1062533

[35] M. Czech, M.-C. Jakobs, and H. Wehrheim. 2015. Just Test What You Cannot Verify!. In *Proc. FASE (LNCS 9033)*. Springer, 100–114. https://doi.org/10.1007/978-3-662-46675-9_7

[36] P. Daca, A. Gupta, and T. A. Henzinger. 2016. Abstraction-Driven Concolic Testing. In *Proc. VMCAI (LNCS 9583)*. Springer, 328–347. https://doi.org/10.1007/978-3-662-49122-5_16

[37] X. Ge, K. Taneja, T. Xie, and N. Tillmann. 2011. DyTa: Dynamic Symbolic Execution Guided with Static Verification Results. In *Proc. ICSE*. ACM, 992–994. https://doi.org/10.1145/1985793.1985971

[38] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. 2010. Compositional May-must Program Analysis: Unleashing the Power of Alternation. In *Proc. POPL*. ACM, 43–56. https://doi.org/10.1145/1706299.1706307

[39] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. 2006. SYNERGY: A New Algorithm for Property Checking. In *Proc. FSE*. ACM, 117–127. https://doi.org/10.1145/1181775.1181790

[40] M. Heizmann, Y.-W. Chen, D. Dietsch, M. Greitschus, A. Nutz, B. Musa, C. Schätzle, C. Schilling, F. Schüssele, and A. Podelski. 2017. Ultimate Automizer with an On-Demand Construction of Floyd-Hoare Automata (Competition Contribution). In *Proc. TACAS (LNCS 10206)*. Springer, 394–398. https://doi.org/10.1007/978-3-662-54580-5_30

[41] M. Heizmann, J. Hoenicke, and A. Podelski. 2013. Software Model Checking for People Who Love Automata. In *Proc. CAV (LNCS 8044)*. Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[42] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. 2004. Abstractions from Proofs. In *Proc. POPL*. ACM, 232–244. https://doi.org/10.1145/964001.964021

[43] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. 2002. Lazy Abstraction. In *Proc. POPL*. ACM, 58–70. https://doi.org/10.1145/503272.503279

[44] M.-C. Jakobs and H. Wehrheim. 2014. Certification for Configurable Program Analysis. In *Proc. SPIN*. ACM, 30–39. https://doi.org/10.1145/2632362.2632372

[45] M.-C. Jakobs and H. Wehrheim. 2015. Programs from Proofs of Predicated Dataflow Analyses. In *Proc. SAC*. ACM, 1729–1736. https://doi.org/10.1145/2695664.2695690

[46] P. Jalote, V. Vangala, T. Singh, and P. Jain. 2006. Program Partitioning: A Framework for Combining Static and Dynamic Analysis. In *Proc. WODA*. ACM, 11–16. https://doi.org/10.1145/1138912.1138916

[47] R. Jhala and R. Majumdar. 2009. Software Model Checking. *Comput. Surveys* 41, 4, Article 21 (2009), 54 pages. https://doi.org/10.1145/1592434.1592438

[48] A. V. Khoroshilov, V. S. Mutilin, A. K. Petrenko, and V. Zakharov. 2009. Establishing Linux Driver Verification Process. In *Proc. Ershov Memorial Conference (LNCS 5947)*. Springer, Berlin, Heidelberg, 165–176. https://doi.org/10.1007/978-3-642-11486-1_14

[49] Y. Köroglu and A. Sen. 2016. Design of a Modified Concolic Testing Algorithm with Smaller Constraints. In *Proc. CSTVA@ISSTA (CEUR 1639)*. CEUR-WS.org, 3–14. http://ceur-ws.org/Vol-1639/paper-03.pdf

[50] A. Lal, S. Qadeer, and S. K. Lahiri. 2012. A Solver for Reachability Modulo Theories. In *Proc. CAV (LNCS 7358)*. Springer, 427–443. https://doi.org/10.1007/978-3-642-31424-7_32

[51] K. Li, C. Reichenbach, C. Csallner, and Y. Smaragdakis. 2014. Residual Investigation: Predictive and Precise Bug Detection. *ACM Transactions on Software Engineering and Methodology* 24, 2 (2014), 7:1–7:32. https://doi.org/10.1145/2656201

[52] G. C. Necula. 1997. Proof-Carrying Code. In *Proc. POPL*. ACM Press, 106–119. https://doi.org/10.1145/263699.263712

[53] H. Post, C. Sinz, A. Kaiser, and T. Gorges. 2008. Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking. In *Proc. ASE*. IEEE, 188–197. https://doi.org/10.1109/ASE.2008.29

[54] Z. Rakamaric and M. Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proc. CAV (LNCS 8559)*. Springer, 106–113. https://doi.org/10.1007/978-3-319-08867-9_7

[55] H. Seo and S. Kim. 2014. How We Get There: A Context-guided Search Strategy in Concolic Testing. In *Proc. FSE*. ACM, 413–424. https://doi.org/10.1145/2635868.2635872

[56] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proc. NDSS*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[57] P. Wendler. 2013. CPAchecker with Sequential Combination of Explicit-State Analysis and Predicate Analysis (Competition Contribution). In *Proc. TACAS (LNCS 7795)*. Springer, 613–615. https://doi.org/10.1007/978-3-642-36742-7_45

[58] D. Wonisch, A. Schremmer, and H. Wehrheim. 2013. Programs from Proofs - A PCC Alternative. In *Proc. CAV (LNCS 8044)*. Springer, 912–927. https://doi.org/10.1007/978-3-642-39799-8_65

[59] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao. 2015. Postconditioned Symbolic Execution. In *Proc. ICST*. IEEE, 1–10. https://doi.org/10.1109/ICST.2015.7102601