FRed: Conditional Model Checking via Reducers and Folders

Dirk Beyer[1] and Marie-Christine Jakobs[2,1]

[1] LMU Munich, Munich, Germany
[2] TU Darmstadt, Department of Computer Science,
Darmstadt, Germany

Abstract. There are many hard verification problems that are currently only solvable by applying several verifiers that are based on complementing technologies. Conditional model checking (CMC) is a successful solution for cooperation between verification tools. In CMC, the first verifier outputs a condition describing the state space that it successfully verified. The second verifier uses the condition to focus its verification on the unverified state space. To use arbitrary second verifiers, we recently proposed a reducer-based approach. One can use the reducer-based approach to construct a conditional verifier from a reducer and a (non-conditional) verifier: the reducer translates the condition into a residual program that describes the unverified state space and the verifier can be any off-the-shelf verifier (that does not need to understand conditions). Until now, only one reducer was available. But for a systematic investigation of the reducer concept, we need several reducers. To fill this gap, we developed FRed, a Framework for exploring different REDucers. Given an existing reducer, FRed allows us to derive various new reducers, which differ in their trade-off between size and precision of the residual program. For our experiments, we derived seven different reducers. Our evaluation on the largest and most diverse public collection of verification problems shows that we need all seven reducers to solve hard verification tasks that were not solvable before with the considered verifiers.

1   Introduction

Due to the undecidability of software verification, even after more than 40 years of research on automatic software verification [31], some hard verification tasks cannot be solved by a single verifier alone. To increase the number of solvable tasks, one needs to combine the strengths of distinct verifiers. Several combinations [3, 8, 9, 20, 23, 25, 32, 33, 37] were proposed in the literature. One promising combination is *conditional model checking* (CMC) [9], which unlike others does not modify the programs nor let the combined techniques know each other.
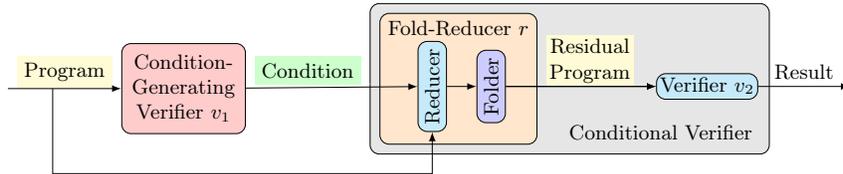
Fig. 1: Reducer-based CMC configuration $(v_2 \circ r) \circ v_1$ with FRED

CMC works as follows: If the first verifier gives up on the verification task, it outputs a condition that describes the state space that it successfully verified. The (conditional) second verifier uses the condition of the first verifier to focus its work on the still-unverified state space. Note that one can easily extend the CMC approach to more than two verifiers by letting all verifiers generate conditions.

To easily construct conditional verifiers (i.e., verifiers that understand conditions) from existing off-the-shelf verifiers, a recent work proposed the concept of reducer-based CMC [13]. Instead of making a verifier aware of conditions, reducer-based CMC constructs a conditional verifier from an existing verifier by plugging a *reducer* in front of the verifier. The reducer is a preprocessor that given the original program and the condition as input, translates the condition into a (residual) program, a format that is understandable by classic verifiers.

The construction of a reducer, especially proving its soundness, is complex and so far there exists only one reducer. However, this reducer's translation is very precise, and therefore, may construct programs that are orders of magnitudes larger than the original program. To solve this problem, and to support systematic experimentation with different reducers, we propose the formal framework FRED, which streamlines and simplifies the construction of new reducers from existing ones. Its underlying idea is to construct a new reducer $r = F \circ R$, a so-called fold reducer, by sequentially composing an existing reducer R with a folder F. A *folder* uses a heuristic that specifies how to modify the program constructed by the existing reducer. More concretely, a folder defines which program locations of the program constructed by the existing reducer are collapsed into a new location and, thus, specifies how to coarsen the program. However, to avoid false alarms, the specified coarsening must not add new program behavior.

New conditional verifiers CV can be constructed with FRED according to the equation $CV = V \circ (F \circ R)$, where $r = (F \circ R)$ is the fold-reducer composed of the existing reducer R and a folder F, V is an arbitrary verifier, and $\circ$ is the sequential composition operator. Figure 1 illustrates this construction in the context of reducer-based CMC. We used this construction to build 49 conditional verifiers, which use the already existing reducer, one of seven folders, and one of seven verifiers. Our large experimental study revealed that using several reducers (with different folders) can make the overall verification more effective.

Contributions. We make the following contributions:

- We introduce FRED, a framework for the composition of new reducers from existing reducers and folding heuristics.
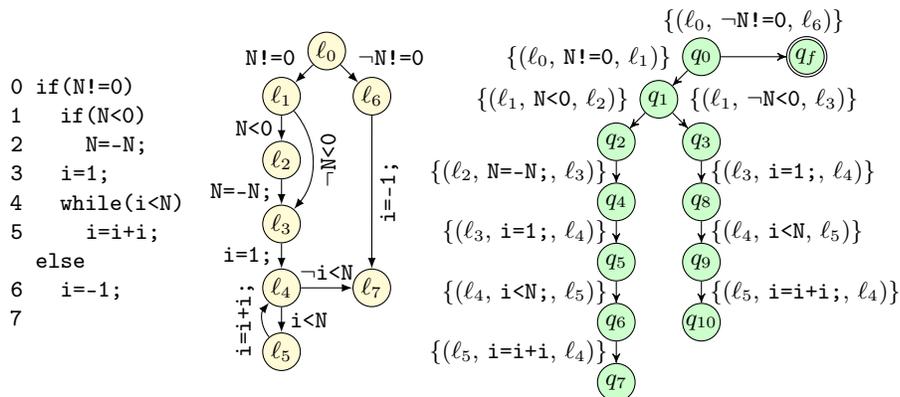
```
0 if(N!=0)
1    if(N<0)
2       N=-N;
3    i=1;
4    while(i<N)
5       i=i+i;
   else
6    i=-1;
7
```

$\{(\ell_0, \neg\texttt{N!=0}, \ell_6)\}$

$\{(\ell_0, \texttt{N!=0}, \ell_1)\}$

$\{(\ell_1, \texttt{N<0}, \ell_2)\}$ $\{(\ell_1, \neg\texttt{N<0}, \ell_3)\}$

$\{(\ell_2, \texttt{N=-N;}, \ell_3)\}$ $\{(\ell_3, \texttt{i=1;}, \ell_4)\}$

$\{(\ell_3, \texttt{i=1;}, \ell_4)\}$ $\{(\ell_4, \texttt{i<N}, \ell_5)\}$

$\{(\ell_4, \texttt{i<N;}, \ell_5)\}$ $\{(\ell_5, \texttt{i=i+i;}, \ell_4)\}$

$\{(\ell_5, \texttt{i=i+i}, \ell_4)\}$

Fig. 2: Example program `absPow`, its CFA, and a condition for our example `absPow` with accepting state q_f and assumptions elided (all true)

- We prove that FRED derives valid reducers in case the existing reducer is valid and the folding heuristic adheres to a correctness constraint.
- We use our framework FRED to derive seven new reducers from the existing reducer PARCOMP [13] and use them in various conditional verifiers.
- We experimentally show that the overall effectiveness of reducer-based CMC can be increased using various reducers.
- Our reducers and all experimental data are available for replication and to construct further conditional model checkers (see Sect. 6).

2   Background

Program Representation. Following the literature [8, 10], we model a program by a *control-flow automaton* (CFA) $C = (L, \ell_0, G)$ consisting of a set L of locations, an initial location $\ell_0 \in L$, and a set of control-flow edges $G \subseteq L \times Ops \times L$. The set Ops describes all possible operations. In our presentation, we only consider operations on integer variables that are either boolean expressions (so called assume operations) or assignments. However, our implementation supports C programs. In the following, we use $\mathcal{L}$ for the superset of all location sets and $\mathcal{C}$ for the set of all CFAs. A *CFA $C = (L, \ell_0, G)$ is deterministic* (i.e., representable as a C program) if for all control-flow edges $(\ell, op_1, \ell_1), (\ell, op_2, \ell_2) \in G$ either $op_1 = op_2$ and $\ell_1 = \ell_2$, or op_1 and op_2 are assume operations with $op_1 \equiv \neg op_2$.

The left of Fig. 2 shows our example program `absPow`, which computes $f(N) = 2^{\lceil \log_2 |N| \rceil}$ for $N \neq 0$ and e.g., ensures the property $f(N) \neq 0$. Next to program `absPow`, its deterministic CFA is shown, which contains one edge per assignment and two edges for each condition of an if- or while-statement. The two edges per if- or while-statement are labeled with the condition and its negation and represent the two evaluations of the condition.

Program Semantics. We use an operational semantics and represent a program's state by a pair of location ℓ (the value of the program counter) and concrete data state c. In our representation, a concrete data state is a mapping from the program variables into the set of integer values. Now, a concrete *path* π *of a CFA* $C = (L, \ell_0, G)$ is a sequence $(\ell_0, c_0) \xrightarrow{g_1} \cdots \xrightarrow{g_n} (\ell_n, c_n)$ such that for all $1 \le i \le n : g_i = (\ell_{i-1}, op_i, \ell_i) \in G$ and $c_{i-1} \xrightarrow{op_i} c_i$, i.e., (a) in case of assume operations, $c_{i-1} \models op_i$ and $c_{i-1} = c_i$ or (b) in case of assignments, $c_i = \mathrm{SP}_{op_i}(c_{i-1})$ and SP is the strongest-post operator of the semantics. We let $paths(C)$ be the set of all concrete paths of a CFA C. Given a concrete path $\pi = (\ell_0, c_0) \xrightarrow{g_1} \cdots \xrightarrow{g_n} (\ell_n, c_n)$, we derive its execution $ex(\pi) = c_0 c_1 \ldots c_n$. Finally, we define $ex(C) := \{ex(\pi) \mid \pi \in paths(C)\}$ to be the executions of a CFA C.

Condition. After an (incomplete) verification run, a condition sums up which concrete paths of a program have been explored [9]. We model the condition as an automaton describing the syntactical program paths that have been verified and the assumptions that have been made on these paths (i.e., which concrete data states were included). Thus, the condition's edges are labeled by pairs of program edges and assumptions. We model assumptions as state conditions, letting Φ denote the set of all state conditions. Accepting states subsume explored paths, i.e., if a path's prefix is accepted by the condition, the path has been explored. Non-explored paths either end in a non-accepting state or more often have a prefix that ends in a state q from which no further transition is applicable. Typically, the latter means that the verifier did not explore beyond the prefix.

The automaton on the right of Fig. 2 shows a condition for our example program `absPow`. For the sake of presentation, we left out the assumptions, which are all true. The condition states that the else-branch of the outermost if-statement was explored and that the verifier performed a BFS alike exploration of the if-branch, which split the exploration of the inner if-branch and which is interrupted after one loop unrolling. Formally, a condition is defined as follows.[3]

Definition 1. *A condition* $A = (Q, \Sigma, \delta, q_0, F)$ *consists of*

- *a finite set* Q *of states, an initial state* $q_0 \in Q$, *and accepting states* $F \subseteq Q$,
- *an alphabet* $\Sigma \subseteq 2^G \times \Phi$, *and*
- *a transition relation* $\delta \subseteq Q \times \Sigma \times Q$ *with* $\neg \exists (q_f, \cdot, q) \in \delta : q_f \in F \wedge q \notin F$.

We let $\mathcal{A}$ *be the set of conditions.*

As already said, a condition describes which paths of a program have been looked at. The following definition formalizes this coverage property. Note that we use $c \models \varphi$ to describe that a concrete data state c satisfies a state condition φ.

Definition 2. *A condition* $A = (Q, \Sigma, \delta, q_0, F)$ *covers a concrete path* $\pi = (\ell_0, c_0) \xrightarrow{g_1} \cdots \xrightarrow{g_n} (\ell_n, c_n)$ *if there exists a run* $\rho = q_0 \xrightarrow{(G_1, \varphi_1)} \ldots \xrightarrow{(G_k, \varphi_k)} q_k$ *in* A *such that (a)* $0 \le k \le n$, *(b)* $q_k \in F$, *and (c)* $\forall 1 \le i \le k : g_i \in G_i \wedge (c_i \models \varphi_i)$.

---

[3] This paper considers only conditions that are represented as automata, while CMC in general [9] is not restricted to a particular representation.

Reducer. The CMC approach suggests that after an incomplete verification run, a second verifier should use the produced condition to explore only the uncovered paths. However, many verifiers do not understand conditions. To overcome this problem, reducer-based CMC [13] suggests to extend verifiers with a preprocessing step that translates the condition into a residual program. A residual program may overapproximate those program paths that are not covered by the condition, but must not introduce additional program behavior. We follow reducer-based CMC [13] and use *reducers* to compute residual programs.

Definition 3. *A* reducer *is a function* $red : \mathcal{C} \times \mathcal{A} \to \mathcal{C}$ *satisfying the residual property:* $\forall C \in \mathcal{C}, \forall A \in \mathcal{A} : ex(C) \setminus \{ex(\pi) \mid A$ *covers* $\pi\} \subseteq ex(red(C, A)) \subseteq ex(C).$

First, a reducer for a specific class of conditions was proposed [26]. Then, reducer-based CMC [13] generalized the first approach to use a reducer, named PARCOMP, which supports all kinds of conditions, and showed that it is indeed a reducer [13]. To compute a residual program, the reducer PARCOMP performs a parallel composition of the program and the condition. Starting in the initial location and initial condition state, it matches CFA edges with condition transitions that subsume the respective CFA edge. If no matching condition transition exists, PARCOMP switches to consider CFA edges only. Additionally, it stops exploring states containing a final state $q \in F$ since the condition covers all longer paths.

However, the reducer PARCOMP has one drawback. Verifiers often unfold the program, e.g., unroll loops or inspect branches separately. Due to partially explored paths, some of the unfoldings become part of the condition and will be encoded in the residual program generated by PARCOMP. Thus, the residual program constructed by PARCOMP may become orders of magnitudes larger than the original program resulting in increased parsing costs for the second verifier. Additionally, a verifier v_2 analyzing the residual program generated by PARCOMP is forced to apply the same unfoldings on the non-covered paths as the condition-generating verifier v_1. However, it might be more effective or efficient if verifier v_2 would less often (or never) unfold certain program structures of the original program. To tackle this problem, we present the framework FRED that extends reducers like PARCOMP to let them compute smaller residual programs with fewer unfoldings at the cost of adding more explored paths to the residual program, i.e., computing less precise residual programs.

3   FRED: **Fold-Reducers from Reducers**

To assist a systematic exploration of the reducer design space, we present the framework FRED. With FRED one can methodically derive new reducers from existing ones, thereby controlling the precision and size of the produced residual programs. One only needs to define how to compress the residual programs computed by the original reducer. Currently, FRED is limited to the class of path-preserving reducers. *Path-preserving reducers* have the advantage that they keep the reference to the original program within the syntactical structure of the residual program, i.e., except for location renaming they encode a subset

of the syntactical paths of the original program. This makes it easier to derive new reducers from them. Next, we formally define a path-preserving reducer, where $\mathcal{U}$ is the universe of location markers (e.g., condition states).

Definition 4. *A reducer ppr is* path-preserving *if for any generated residual program $ppr((L, \ell_0, G), A) = (L_r, \ell_{0,r}, G_r)$ it is valid that (a) $L_r \subseteq L \times \mathcal{U}$ for some $\mathcal{U}$, (b) $\ell_{0,r} = (\ell_0, \cdot)$, and (c) $\forall((\ell, u), op, (\ell', u')) \in G_r : \exists(\ell, op, \ell') \in G$.*

Given a path-preserving reducer like PARCOMP, the goal of FRED is to derive new reducers that produce smaller, less precise residual programs. Our idea is that the new reducers aggregate certain similar behavior of the residual program C_r produced by the given path-preserving reducer. So far, the framework FRED supports syntactical aggregations that unite location states of the program C_r. These aggregations can be used to revert loop-unfoldings or separation of branches, the main cause for large residual programs. Additionally, these aggregations are simple to compute. One needs to define only a partitioning of C_r's location states into equivalence classes. However, to get proper reducers, the derived reducers must not introduce new program behavior. Transferred to our aggregations, this means that we must not combine location states of C_r that refer to different locations of the original program. We introduce the concept of a *location-consistent partitioner* that computes partitions respecting this requirement.

Definition 5. *A* location-consistent partitioner *is a function p that maps a set $L_r \subseteq \mathcal{L} \times \mathcal{U}$ to a partition $\{L_1, \ldots, L_n\}$ of L_r s.t. $\forall 1 \leq i \leq n : |\{\ell \mid (\ell, \cdot) \in L_i\}| = 1$. We use $\mathcal{P}$ for the set of all location-consistent partitioners.*

As examples, we consider the two extreme location-consistent partitioners cfa and sep as defined in the following. Partitioner cfa groups all elements with the same location and sep never groups elements.

$$\mathrm{cfa}(L_r) = \left\{ \{(\ell, u) \in L_r \mid \ell = \ell'\} \mid \exists(\ell', \cdot) \in L_r \right\} \quad \mathrm{sep}(L_r) = \left\{ \{(\ell, u)\} \mid (\ell, u) \in L_r \right\}$$

All remaining location-consistent partitioners group subsets of elements with same locations. Often, they are context dependent, i.e., they take into account the structure of the original program or the program C_r generated by the path-preserving reducer. For instance, we use the following partitioner that combines locations referring to the same loop head in the original program. The partitioner is parameterized by the loop heads L' of the original program.

$$\mathrm{lh}_{L'}(L_r) = \mathrm{cfa}\big(\{(\ell, u) \in L_r \mid \ell \in L'\}\big) \ \cup \ \mathrm{sep}\big(\{(\ell, u) \in L_r \mid \ell \notin L'\}\big)$$

A partioning of the nodes of a graph, e.g., a CFA, induces a coarser graph. Each set of nodes becomes a node of the new graph and there exists an edge between two sets of nodes if there exists an edge between two nodes in the original graph, one in each set. A *folder* applies this principle to compress a residual program computed by a path-preserving reducer. A location-consistent partitioner defines the partitioning of location states. Furthermore, the new initial program location is the set of location states that contains the original initial location. Due to the partitioner's properties, exactly one such set exists.

```
0 if(N!=0)    0 if(N!=0)    0 if(N!=0)     0 if(N!=0)     0  if(N!=0)
1  if(N<0)    1  if(N<0)    1  if(N<0)     1  if(N<0)     1   if(N<0)
2   N=-N;     2   N=-N;     2   N=-N;      2   N=-N;      2    N=-N;
3  i=1;       3  i=1;       3  i=1;        3  i=1;        3   i=1;
4  while(i<N)    else       4  if(i<N)        else        4   if(i<N)
5   i=i+i;    4   i=1;      5   i=i+i;     4   i=1;       5    i=i+i;
              5  while(i<N) 6  while(i<N)  5  if(i<N)         else
              6   i=i+i;    7   i=i+i;     6   i=i+i;     7   i=1;
                                          7  while(i<N)   8   if(i<N)
                                          8   i=i+i;      9    i=i+i;
                                                          10  while(i<N)
                                                          11   i=i+i;
```

(a) CFA   (b) LH, LHC   (c) NLH   (d) LHB   (e) LHBC, SEP

Fig. 3: Five residual programs with increasing program sizes and varying program structure, constructed by the seven fold-reducers considered in the evaluation

Definition 6. *A* folder fold $: \mathcal{C} \times \mathcal{P} \to \mathcal{C}$ *compresses a CFA* $C_r = (L_r, \ell_{0,r}, G_r)$ *with a location-consistent partitioner* p *such that*

$$\text{fold}((L_r, \ell_{0,r}, G_r), p) = (p(L_r), \ell_{0,p}, G_p) \text{ with}$$

$\ell_{0,r} \in \ell_{0,p}$ *and* $G_p = \big\{ (\ell_p, op, \ell'_p) \,\big|\, \ell_p, \ell'_p \in p(L_r) \wedge \exists (\ell, op, \ell') \in G_r : \ell \in \ell_p \wedge \ell' \in \ell'_p \big\}$.

We use folders to construct so called *fold-reducers* from an existing path-preserving reducer. To this end, we concatenate the path-preserving reducer with a folder.

Definition 7. *Let* p *be a location-consistent partitioner and* ppr *a path-preserving reducer. The* fold-reducer *for* p *and* ppr *is* $\text{FOLDRED}_p^{ppr}(C, A) := \text{fold}(ppr(C, A), p)$.

Figure 3 shows five residual programs constructed from program absPow (Fig. 2, left) and the condition for it (Fig. 2, right). The residual programs differ in their program size and structure. They were constructed by the seven different fold-reducers used in the evaluation, all of them using the reducer PARCOMP [13], but we converted them into a better readable form using proper if- and while-statements instead of gotos. Note that for this example, some fold-reducers constructed the same residual program. To construct the residual programs in Figs. 3a and 3e, the partitioners cfa and sep could be used, respectively. For the residual program in Fig. 3b, we used partitioner $\text{lh}_{L'}$ with $L' = \{\ell_4\}$. The partitioner used to construct the program in Fig. 3c undoes unfoldings of if-statements but keeps loop-unfoldings. Finally, the program in Fig. 3d is generated with a partitioner that allows loop-unfoldings up to a given bound of ten and then folds them. However, loop heads of the same iteration are always combined.

Above, we used fold-reducers to compute residual programs. In general, we plan to use fold-reducers in the construction of conditional verifiers. Thus, we must show that fold-reducers are reducers. Syntactically, fold-reducers look like reducers. It remains to be shown that fold-reducers fulfill the residual property.

Theorem 1. *Every fold-reducer* $\mathrm{FOLDRED}_p^{ppr}$ *is a reducer.*

Proof. We need to show that $\mathrm{ex}(C)\backslash\{\mathrm{ex}(\pi) \mid A \text{ covers } \pi\} \subseteq \mathrm{ex}(\mathrm{FOLDRED}_p^{ppr}(C,A))$ $\subseteq \mathrm{ex}(C)$. Since ppr is reducer, $\mathrm{ex}(C) \setminus \{\mathrm{ex}(\pi) \mid A \text{ covers } \pi\} \subseteq \mathrm{ex}(ppr(C,A))$. Thus, it suffices to show that $\mathrm{ex}(ppr(C,A)) \subseteq \mathrm{ex}(\mathrm{FOLDRED}_p^{ppr}(C,A)) \subseteq \mathrm{ex}(C)$.

In the following, let $C = (L_o, \ell_{0,o}, G_o)$, $ppr(C,A)) = (L_r, \ell_{0,r}, G_r)$, and $\mathrm{FOLDRED}_p^{ppr}(C,A) = (L_f, \ell_{0,f}, G_f)$. Due to the requirements on p and the definition of the fold-reducer, there exists a unique function $h : L_r \to L_f$ with $\forall \ell_r \in L_r : \ell_r \in h(\ell_r)$ and $h(\ell_{0,r}) = \ell_{0,f}$.

Part I) $\mathrm{ex}(ppr(C,A)) \subseteq \mathrm{ex}(\mathrm{FOLDRED}_p^{ppr}(C,A))$:

$\qquad c_0 c_1 \ldots c_n \in \mathrm{ex}(ppr(C,A))$

$\Rightarrow$ there exists $\pi_r = (\ell_{0,r}, c_0) \xrightarrow{g_1^r} \cdots \xrightarrow{g_n^r} (\ell_{n,r}, c_n)$ s.t.
$\qquad \forall 1 \le i \le n : g_i^r = (\ell_{i-1,r}, op_i, \ell_{i,r}) \in G_r \wedge c_{i-1} \xrightarrow{op_i} c_i$

$\Rightarrow \forall 1 \le i \le n : \exists g_i^f = (h(\ell_{i-1,r}), op_i, h(\ell_{i,r})) \in G_f$

$\Rightarrow \pi_f = (h(\ell_{0,r}), c_0) \xrightarrow{g_1^f} \cdots \xrightarrow{g_n^f} (h(\ell_{n,r}), c_n)$ is a concrete path
$\qquad$ of $\mathrm{FOLDRED}_p^{ppr}(C,A)$

$\Rightarrow c_0 c_1 \ldots c_n \in \mathrm{ex}(\mathrm{FOLDRED}_p^{ppr}(C,A))$

Part II) $\mathrm{ex}(\mathrm{FOLDRED}_p^{ppr}(C,A)) \subseteq \mathrm{ex}(C)$:

$\qquad c_0 c_1 \ldots c_n \in \mathrm{ex}(\mathrm{FOLDRED}_p^{ppr}(C,A))$

$\Rightarrow$ there exists $\pi_f = (\ell_{0,f}, c_0) \xrightarrow{g_1^f} \cdots \xrightarrow{g_n^f} (\ell_{n,f}, c_n)$ s.t.
$\qquad \forall 1 \le i \le n : g_i^f = (\ell_{i-1,f}, op_i, \ell_{i,f}) \in G_f \wedge c_{i-1} \xrightarrow{op_i} c_i$

$\Rightarrow \forall 1 \le i \le n$ there exists $g_i^r = (\ell_{i,r}, op_i, \ell'_{i,r}) \in G_r$
$\qquad$ with $\ell_{i,r} \in \ell_{i-1,f} \wedge \ell'_{i,r} \in \ell_{i,f}$

$\Rightarrow \forall 1 \le i \le n$ there exists $g_{i,o} = (\ell_{i,o}, op_i, \ell'_{i,o}) \in G_o$
$\qquad$ with $\ell_{i,r} = (\ell_{i,o}, \cdot) \wedge \ell'_{i,r} = (\ell'_{i,o}, \cdot)$

$\Rightarrow \forall 1 \le i \le n : (i = 1 \vee \ell_{i,o} = \ell'_{i-1,o}) \wedge \ell_{1,o} = \ell_{0,o}$
$\qquad$ (since $\ell'_{i,r}, \ell_{i+1,r} \in \ell_{i,f}, \ell_{1,r} \in \ell_{0,f}$, p location-consistent)

$\Rightarrow (\ell_{1,o}, c_0) \xrightarrow{g_{1,o}} (\ell'_{1,o}, c_1) \xrightarrow{g_{2,o}} \cdots \xrightarrow{g_{n,o}} (\ell'_{n,o}, c_n) \in path(C)$

$\Rightarrow c_0 c_1 \ldots c_n \in \mathrm{ex}(C)$

In practice, arbitrary fold-reducers are unsatisfactory since they may produce non-deterministic CFAs, which cannot be translated to C programs. Figure 4 shows an example of a non-deterministic CFA generated by a fold-reducer. In the example, the non-determinism is caused by the partitioner $lh_{\{\ell_4\}}$, which only combines loop heads. Generally, also the condition may cause non-determinism.[4] To solve the non-determinism problem, we transform a fold-reducer into a deterministic fold-reducer that generates deterministic residual programs from deterministic, input programs. The basic idea is to adapt the partitioner to compute a coarser

---

[4] Theoretically, the non-determinism may also be caused by a non-deterministic, original program. However, we assume that the original program is deterministic.
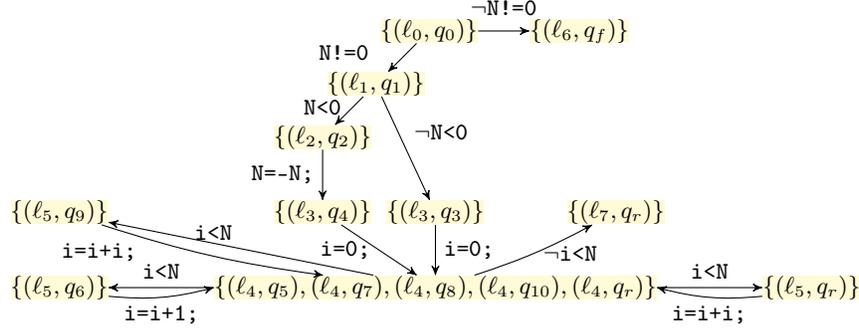
Fig. 4: Nondeterministic residual program built from program `absPow`, the condition from Fig. 2, and a fold-reducer using reducer PARCOMP and partitioner $lh_{\{\ell_4\}}$

Algorithm 1 det

Input: CFA $C_r = (L_r, \ell_{0,r}, G_r)$, p // residual program, location-consistent partitioner

Output: $part$ // location-consistent partition of L_r
1: oldPart:=$\emptyset$; part:=$p(L_r)$;
2: **while** oldPart $\neq$ part **do**
3:     oldPart:=part;
4:     **for** each $(L_i, L_j, L_k) \in$ oldPart $\times$ part $\times$ part **do**
5:         **if** $L_i \in$ part $\wedge L_i \neq L_j \wedge \exists(\ell_k, op, \ell_i), (\ell'_k, op, \ell_j) \in G_r$:
            $\ell_k, \ell'_k \in L_k \wedge \ell_i \in L_i \wedge \ell_j \in L_j$ **then**
6:             part:=(part$\setminus\{L_i, L_j\}) \cup \{L_i \cup L_j\}$;
7: **return** part

---

partitioning. The coarser partitioning combines all partition elements of the original partition that would cause the residual program to be non-deterministic.

Algorithm 1 shows how to compute such a coarser partitioning from the original partitioning. Starting with the original partitioning, it combines partitions of its current partitioning as long as there exist two CFA edges causing nondeterminism, i.e., they consider the same operation and start in the same partition element, but end in different partition elements.

Attentive readers already noticed that Alg. 1 uses the program C_r generated by the path-preserving reducer to adapt the partitioning. Since multiple programs may consider the same set of location states but different control-flow edges, it is impossible to adapt the partitioner without knowledge of C_r. Thus, a deterministic fold-reducer must use different adaptions of the partitioner p. The correct adaption depends on the input program and the path-preserving reducer. We use the following adaption, which depends on the original program and the path-preserving reducer ppr used by the fold-reducer.

$$\det_{ppr(C,A),p}(L) := \begin{cases} \det(ppr(C,A), p) & \text{if } ppr(C,A) = (L, \cdot, \cdot) \wedge C \text{ deterministic} \\ p(L) & \text{else} \end{cases}$$

The adapted partitioner returns the partitioning computed by the original partitioner except for one case. When the original program C is deterministic and the adapted partitioner is given the location states of the program computed by the path-preserving reducer, the partition is adapted with Alg. 1. Note that we neglect to apply Alg. 1 for non-deterministic original programs, because it then may combine partitions considering different location states of the original program, thus, resulting in a location-inconsistent partitioner. However, to use the adapted partitioner in a fold-reducer, it must remain location-consistent.

Lemma 1. *For a given CFA C, condition A, path-preserving reducer ppr, location-consistent partitioner p, function $\det_{ppr(C,A),p}$ is a location-consistent partitioner.*

Knowing that the adapted partitioner remains location-consistent, we explain how to derive a *deterministic fold-reducer* from a fold-reducer. The idea is simple. The deterministic fold-reducer uses for each input program a dedicated variant of the original fold-reducer. This dedicated variant uses the prescribed adaption $\det(ppr(C,A),p)$ of the original partitioner to the original program.

Definition 8. *Let $\mathrm{FOLDRED}_p^{ppr}$ be a fold-reducer. We define the* deterministic fold-reducer *to be $\mathrm{FOLDRED}_{p,ppr}^{\det}(C,A) := \mathrm{FOLDRED}_{\det_{ppr(C,A),p}}^{ppr}(C,A)$.*

We already showed that the proposed adaption of the location-consistent partitioner results in a location-consistent partitioner. Now, we can easily conclude that deterministic fold-reducers guarantee the residual property and, thus, can be used to construct conditional verifiers.

Corollary 1. *Every deterministic fold-reducer $\mathrm{FOLDRED}_{p,ppr}^{\det}$ is a reducer.*

While the previous property is mandatory, we build deterministic fold-reducers to produce deterministic programs when given deterministic programs. The subsequent proposition certifies this property of deterministic fold-reducers.

Proposition 1. *Given a deterministic fold-reducer $\mathrm{FOLDRED}_{p,ppr}^{\det}$, a deterministic CFA C, and a condition A, then the residual program $\mathrm{FOLDRED}_{p,ppr}^{\det}(C,A)$ is deterministic.*

4   Evaluation

The main goals of our experiments are to systematically investigate different (fold-)reducers and to find out whether fold-reducers can overcome the problem that reducer PARCOMP sometimes generates too large and precise residual programs. Since PARCOMP was the only available reducer our goal was to counteract on its weaknesses (i.e., the sometimes large residual programs), investigating whether one needs to settle for PARCOMP's weakness is beyond the scope of this evaluation. Another goal of our evaluation is to compare CMC with fold-reducers against non-cooperative combinations, especially sequential combinations. This leads us to three research questions:

RQ 1. Do distinct fold-reducers generate different residual programs?

RQ 2. Can fold-reducer be better than reducer ParComp and is there a reducer that dominates the others?

RQ 3. Can reducer-based CMC replace non-cooperative verifier combinations?

4.1 Experimental Setup

CMC Configurations. A reducer-based CMC configuration consists of (1) a condition-generating verifier v_1, (2) a reducer r, and (3) a second verifier v_2 (cf. Fig. 1). For components v_1 and r, we use CPAchecker [14] in revision r32965 since it already provides condition-generating verifiers and reducer ParComp [13].

As in other works [9, 13], we use a predicate analysis [15] and a value analysis [16], both using a time limit of $100\,\text{s}^5$, as condition-generating verifiers. If they do not succeed within 100 s, they give up and output a condition. For verifier v_2, we use the three tools CPA-Seq [29], ESBMC [34], and VeriAbs [30] that performed best on the reachability categories of SV-COMP 2020^6 as well as Symbiotic, which performed best in the SoftwareSystems category of SV-COMP 2020. For all four tools, we use their version submitted to SV-COMP 2020. Additionally, we used three well-maintained analyses, kInduction [7], predicate analysis [15], and value analysis [16], which are part of the award-winning sequential composition of CPAchecker [29]. For them, we also use CPAchecker revision r32965.

We investigated seven fold-reducers r, which we implemented in the FRed plug-in for CPAchecker. All fold-reducers inline functions and typically use the deterministic fold-reducer variant of the reducers described in Sect. 3. Only the CFA and the SEP reducers already generate deterministic, residual programs and do not need to use the deterministic variant. The seven fold-reducers are:

CFA Fold-reducer that uses partitioner *cfa*, i.e., it combines elements with same location states and, thus, reconstructs those parts of the original CFA that have not been fully explored.

LH Fold-reducer that is based on partitioner $lh_{L'}$ and undoes loop-unfoldings. It combines all elements with the same loop-head location state from L'.

LHC Fold-reducer that also aims at reverting loop-unfoldings, but avoids to combine loop executions started in different contexts, i.e., reached on different syntactical paths ignoring finished loops.

LHB Fold-reducer that limits loop-unfoldings, i.e., keeps loop-unfoldings up to a given bound (we use 10) and afterwards collapse the unfoldings.

LHBC Fold-reducer that like LHB limits loop-unfoldings up to a bound of 10, but additionally separates loop executions with different contexts like LHC.

NLH Fold-reducer that undoes branch- but not loop-unfoldings (keeps different loop iterations separated).

SEP Fold-reducer that never combines elements, uses partitioner *sep* (same as ParComp [13]).

Combining each fold-reducer r with all second verifiers v_2 we obtain 49 conditional verifiers $v_2 \circ r$. Combining the conditional verifiers with the condition-generating verifier gives us 84 reducer-based CMC configurations.[7]

Tasks. For our evaluation, we considered the well-established benchmark set[8] from the competition on software verification [4]. We focused on the 6 907 tasks of the ReachSafety categories, because all considered analyses can verify the property "no call to function `__VERIFIER_error()` is reachable". For each condition-generating verifier v_1, we created a task set that excludes all tasks for which all reducers reported an error ($\approx 11\%$) as well as all easy tasks ($\approx 45\%$). A task is considered easy if it does not require CMC because it can be solved in 100 s by v_1 or in 1 000 s[9] by all verifiers v_2. Thus, we only look at tasks for which CMC can contribute additional value (2 949 tasks for CMC with v_1=predicate analysis and 3 046 tasks for CMC with v_1=value analysis).

Execution Framework. We performed our experiments on machines with 33 GB of memory and an Intel Xeon E3-1230 v5 CPU (8 processing units and a frequency of 3.4 GHz). The machines run a Ubuntu 18.04 operating system (Linux kernel 4.15). We use BenchExec [17] to run our experiments. To ensure that all CMC configurations with the same verifier v_1 use the same conditions, we run the condition-generating verifiers v_1 once with a runtime limit of 100 s[10] and a memory limit of 15 GB. The generated conditions are then used when running the conditional verifiers with a runtime limit of 900 s and a memory limit of 15 GB.

Replication Support. Our experimental data are available online (see Sect. 6).

4.2 Experimental Results

RQ 1 (Different residual programs?) Already our example (Fig. 3) shows that residual programs generated by different reducers can significantly differ in the program size and the branching structure. To further investigate the difference of residual programs, we searched our tasks for programs for which all seven reducers generated residual programs with different numbers of program locations, and selected the program `sqrt_Householder_interval.c`. Figure 5 shows graph shapes of the CFAs of the residual programs generated by the seven fold-reducers. In a graph shape, the width of line i is proportional to the number of CFA nodes with a shortest path of length i from the initial location. We observe that the graph shapes differ in their height and width. Thus, residual programs differ in their branching structure. Finally, we looked at the size increase of the residual programs, i.e., number of locations of residual program ($|L_{\mathrm{residual}}|$) divided by number of locations of original program ($|L_{\mathrm{original}}|$). Figure 6 shows boxplots

---

[7] We excluded the 14 combinations in which verifiers v_1 and v_2 are identical because they do not describe a cooperation between different verifiers, but are basically identical to a verification with a single verifier with some additional overhead.

[8] https://github.com/sosy-lab/sv-benchmarks/tree/svcomp20

[9] We grant CMC 1 000 s. We use a a standard time limit of 900 s for the conditional verifier and, as already explained, 100 s for the condition-generating verifier v_1.

[10] To not interrupt condition writing, we applied the limit to the verification algorithm. Imprecise enforcement or condition writing may result in runtimes larger than 100 s.
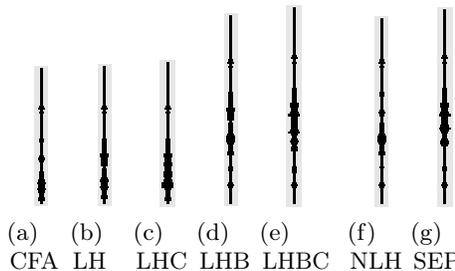
Fig. 5: Shape graphs (indicating structure) of residual programs constructed from program `sqrt_Householder_interval.c` by respective fold-reducer
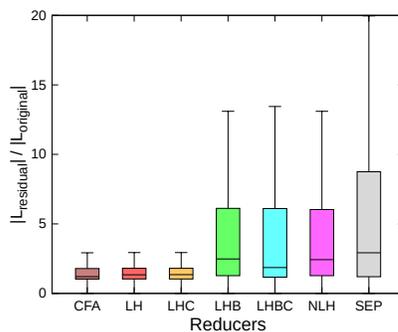

Fig. 6: Boxplot for size increase of residual programs

Table 1: Number of verification tasks solved correctly by each CMC configuration that uses the predicate analysis (upper part) or the value analysis (lower part) for condition generation; last column combines the previous columns

					Verifier v_2								
r	+CPASeq		+ESBMC		+Symbiotic		+VeriAbs		+kInd.		+Val.		+All	
	2949	1636	2949	1773	2949	1983	2949	730	2949	1928	2949	2082	2949	433
CFA	946	28	397	165	541	41	**397**	**24**	762	62	**296**	**17**	**1342**	38
LH	**951**	**30**	397	165	542	42	391	**24**	**764**	64	295	**17**	**1342**	38
LHC	949	**30**	397	165	541	41	395	**24**	761	63	295	**17**	1338	37
LHB	700	29	413	189	510	**43**	365	20	624	68	172	**17**	1129	41
LHBC	699	**30**	412	189	509	**43**	366	20	623	68	169	**17**	1122	41
NLH	722	27	447	212	508	42	367	22	634	**78**	169	**17**	1155	41
SEP	662	29	**500**	**226**	**570**	**43**	**397**	22	614	75	132	16	1195	**42**
All	997	41	558	277	609	46	479	26	783	76	298	17	1501	54
ID	1269	0	1003	0	709	0	2166	0	860	0	657	0	2446	0

					Verifier v_2								
r	+CPASeq		+ESBMC		+Symbiotic		+VeriAbs		+kInd.		+Pred.		+All	
	3046	1713	3046	1800	3046	2112	3046	758	3046	1610	3046	2123	3046	434
CFA	**1018**	**51**	**492**	**178**	**536**	70	**600**	41	**937**	78	697	114	**1452**	**52**
LH	955	49	481	176	515	72	573	41	870	77	682	115	1411	**52**
LHC	940	**51**	481	176	512	72	568	**42**	860	78	683	115	1402	**52**
LHB	822	44	458	143	511	**75**	503	41	761	73	677	**147**	1348	44
LHBC	824	45	458	143	508	**75**	499	41	754	75	674	144	1342	44
NLH	940	**51**	401	90	460	72	549	39	859	**81**	**715**	146	1310	44
SEP	610	43	488	123	410	**75**	440	36	588	70	509	124	957	35
All	1041	64	650	273	548	75	623	45	944	84	772	177	1525	61
ID	1228	0	1045	0	734	0	2210	0	1121	0	673	0	2482	0

depicting for each reducer the distribution of the size increases of its residual programs. We observe that the boxes differ in size, the median (middle line) and the whiskers, which supports that residual programs from distinct reducers differ.

RQ 2 (Better than PARCOMP and existence of dominating reducer?) To answer this research question, we study the number of tasks solved correctly

by the CMC configurations. We focus on correctly solved tasks and exclude incorrectly solved tasks, which are an unreliable source of information caused by an unsound CMC configuration, e.g., due to an unsound verifier or a bug in one of the CMC configurations. For each CMC configuration, we report the numbers for the full task set[11] and for a restricted task set that only considers those tasks that cannot be solved by the two verifiers in the CMC configuration and, thus, requires cooperation, e.g., via CMC. Table 1 shows the numbers for the CMC configurations using the predicate analysis (upper part) and the value analysis (lower part) for the condition-generating verifier v_1. The total number of tasks considered in each column are reported at the top. The CMC configurations are fixed by the reducer (row) and the verifier v_2 (columns). Column '+All' displays the numbers of correctly solved tasks by CMC configurations with any verifier v_2, but excluding tasks that one of the CMC configurations solved incorrectly.[12] Similarly, row 'All' uses any reducer. The last row is discussed later.

Looking at Table 1, we first observe that there exist verifier combinations for which the CMC configurations using the SEP reducer, which is identical to reducer PARCOMP, does not solve the most tasks (bold numbers). We also observe that for some CMC configurations the best reducer differs when considering the full or the restricted task set. Also, the best reducers differ when changing the condition-generating verifier. Hence, the best reducer depends on (1) the task set, and (2) the verifier combination. Additionally, we observe that the numbers in row 'All' are often larger than in the previous rows. Thus, we are more effective when using different reducers. Moreover, our raw data revealed that for all seven reducers there exist tasks that can only be solved by a verifier combination when using this particular reducer. Therefore, we need all seven reducers.

RQ 3 (Replacement for non-cooperative verifier combinations?) To answer this question, we compare CMC with fold-reducers against a combination that executes verifier v_1 and v_2 in sequence using the same program for both verifiers and without exchanging any information. This combination is identical to CMC with the identity reducer ID, which returns the input program. Row ID in Table 1 shows the number of tasks solved correctly by the sequential composition. Obviously, the sequential composition does not solve any task in the restricted task set, which only contains tasks that cannot be solved by v_1 and v_2. To solve these tasks, one needs cooperation approaches like reducer-based CMC. For the full task set, we observe that except for one case row ID solves more tasks than the other rows. Hence, reducer-based CMC should only be used for hard verification tasks that cannot be solved by single verifiers and, thus, need cooperation.

4.3   Threats to Validity

In theory, our reducers fulfill the residual condition. However, in practice our reducer implementation might contain bugs that lead to residual programs that

---

[11] Remember that the full task set depends on the condition-generating verifier v_1 because we only look at tasks for which CMC can contribute additional value.

[12] For +All, the tasks in the restricted set are neither solved by v_1 nor any verifier v_2.

add or miss program behavior, i.e., violate the residual condition. In principle, such bugs can lead to residual programs fulfilling the same property as the original program, but that are easier to verify. Hence, some of the correctly solved tasks might come from such bugs. Furthermore, our results concerning the reducers may not generalize. First, we considered a subset of the SV-COMP tasks and analyses that are run in SV-COMP. The analyses are likely trained on the tasks. However, also CMC configurations that unfold the original program a lot, and thus generate residual programs that look differently from the original program, solved many tasks. We are confident that our results apply to other programs. Second, we used specific time limits for the condition-generating verifier v_1 and the conditional verifier (reducer plus verifier v_2). While we chose common time limits, our results may look differently when using different limits.

5   Related Work

Our work is based on the idea of conditional model checking (CMC) [9], which combines analyses via condition passing. The early conditional model checkers [9] used the condition to directly steer the exploration of the second analysis. Translating the condition into a residual program was first proposed in 2015 [26]. Besides slicing, they construct the residual program from a parallel combination of condition and program. Recently, reducer-based CMC [13] generalized the idea of residual programs and introduced the concept of a reducer. The proposed reducer was similar to the earlier parallel combination [26]. In this paper, we construct multiple, new reducers from the original reducer [13].

Combination of Analyses. One type of combination testifies verification results. These combinations try to confirm alarms [18, 25, 28, 35, 44, 47] or proofs [1, 39, 41, 45], possibly excluding unconfirmed results. Violation and correctness witnesses [5, 6] provide a tool-independent exchange format for alarms and proofs, enabling other tools to check a verifier's result. Further combinations join forces of different analyses. On the one hand, analysis domains are integrated [8, 10, 23, 24, 33] to get more precise domains than the pure product. On the other hand, interleavings of analysis algorithms are proposed [3, 27, 36, 37] to benefit from (intermediate) results of other algorithms. A third class of combinations distributes the verification effort among different tools. CMC [9] and reducer-based CMC [13], which we apply, belong to this class. Often, the program parts that could not be verified by the first analyzer are encoded with programs. Sometimes annotations (assertions) are added [19, 20, 21, 46], while program trimming [32] adds assume statements to the original program. Reducer-based CMC [13] and program partitioning [43] output a new program describing a subset of the original program paths. Abstraction-driven concolic testing [27] interleaves concolic testing and predicate abstraction to construct test cases for test goals. CoVeriTest [11] recently generalized this approach. Conditional static analysis [49] splits the program paths into subsets, runs one dataflow analysis on each subset and finally combines the results of these restricted analyses.

Program Transformation for Verification. Our work uses fold-reducers to transform the original program to remove already-verified paths. Like any reducer, fold-reducers may unfold the structure (execution paths) of the original program. Moreover, fold-reducers use a folder that aims at reverting some of the unfoldings introduced by the existing reducer used in the fold-reducer. Likewise, verification refactoring [53] heuristically undoes compiler optimizations to ease verification. Programs-from-proofs [42] pursues the same goal, but it unfolds the program structure to ease verification. Program partitioning [43] and abstraction-driven concolic testing [27] transform the original program to remove tested or infeasible program paths. Unfolding the program structure is a common approach to remove infeasible paths [2, 38, 48] or improve the analysis result [40, 50, 51]. In contrast, folding is used less often. Examples are compiler optimizations like constant propagation [52] and common-subexpression elimination [22].

6   Conclusion

One solution to the problem of verifying complex software systems is to improve verification algorithms and theories. An orthogonal solution is to combine existing techniques. Conditional model checking (CMC) is a promising approach to combine the strengths of different verifiers. To construct new conditional model checkers from existing model checkers in an implementation-less and configurable manner (off-the-shelf, plug-and-play), the concept of reducer-based CMC was recently proposed [13]. Instead of spending developer resources on adapting existing verifiers to make them understand conditions—the information exchange format in CMC—, reducer-based CMC suggests to put reducers in front of existing, off-the-shelf verifiers. The task of a reducer is to convert the condition into a format that the verifier already understands, namely program code. Until now, only one reducer existed. Our experiments revealed that there is a lot of potential for improving the effectiveness by using different kinds of reducers.

Developing new reducers can be a laborious task. One must define how to compute the residual program from the input condition and program. Moreover, one must prove that the reducer fulfills the residual property, a correctness property for the reducer. To systematically study reducers, we developed the framework FRED, which simplifies the development of new reducers. FRED allows us to derive the new reducer from an existing one and a heuristic that describes how to coarsen the residual program generated by the existing reducer. To prove that the derived reducer is indeed a reducer, one only needs to show that the specified heuristic is a location-consistent partitioner, a property much simpler than the residual property. Our experience with FRED is that developing and implementing a new heuristic takes at most a few hours. In the future, algorithm selection could be applied to choose the most suitable reducer for a task.

Data Availability Statement. The reducers and all experimental data are publicly available for replication on a web page [13] and as replication package [12].

---

[13] https://www.sosy-lab.org/research/fred/

References

1. Albert, E., Puebla, G., Hermenegildo, M.V.: Abstraction-carrying code. In: Proc. LPAR. pp. 380–397. LNCS 3452, Springer (2004). https://doi.org/10.1007/978-3-540-32275-7_25
2. Balakrishnan, G., Sankaranarayanan, S., Ivancic, F., Wei, O., Gupta, A.: SLR: path-sensitive analysis through infeasible-path detection and syntactic language refinement. In: Proc. SAS. pp. 238–254. LNCS 5079, Springer (2008). https://doi.org/10.1007/978-3-540-69166-2_16
3. Beckman, N., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: Proc. ISSTA. pp. 3–14. ACM (2008). https://doi.org/10.1145/1390630.1390634
4. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2). pp. 347–367. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_21
5. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
6. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
7. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
8. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
9. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). https://doi.org/10.1145/2393596.2393664
10. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE. pp. 29–38. IEEE (2008). https://doi.org/10.1109/ASE.2008.13
11. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
12. Beyer, D., Jakobs, M.C.: Replication package for article 'FRed: Conditional model checking via reducers and folders' in Proc. SEFM 2020 (2020). https://doi.org/10.5281/zenodo.3953565
13. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
14. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
15. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010)
16. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11

17. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y

18. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: Proc. SAC. pp. 1284–1291. ACM (2012). https://doi.org/10.1145/2245276.2231980

19. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Proc. FM. pp. 132–146. LNCS 7436, Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_13

20. Christakis, M., Müller, P., Wüstholz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Proc. ICSE. pp. 144–155. ACM (2016). https://doi.org/10.1145/2884781.2884843

21. Christakis, M., Wüstholz, V.: Bounded abstract interpretation. In: Proc. SAS. pp. 105–125. LNCS 9837, Springer (2016). https://doi.org/10.1007/978-3-662-53413-7_6

22. Cocke, J.: Global common subexpression elimination. In: Proc. Symposium on Compiler Optimization. pp. 20–24. ACM (1970). https://doi.org/10.1145/800028.808480

23. Cousot, P., Cousot, R.: Systematic design of program-analysis frameworks. In: Proc. POPL. pp. 269–282. ACM (1979). https://doi.org/10.1145/567752.567778

24. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the ASTRÉE static analyzer. In: Proc. ASIAN'06. pp. 272–300. LNCS 4435, Springer (2008). https://doi.org/10.1007/978-3-540-77505-8_23

25. Csallner, C., Smaragdakis, Y.: Check 'n' crash: Combining static checking and testing. In: Proc. ICSE. pp. 422–431. ACM (2005). https://doi.org/10.1145/1062455.1062533

26. Czech, M., Jakobs, M., Wehrheim, H.: Just test what you cannot verify! In: Proc. FASE. pp. 100–114. LNCS 9033, Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_7

27. Daca, P., Gupta, A., Henzinger, T.A.: Abstraction-driven concolic testing. In: Proc. VMCAI. pp. 328–347. LNCS 9583, Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_16

28. Dams, D., Namjoshi, K.S.: Orion: High-precision methods for static error analysis of C and C++ programs. In: Proc. FMCO. pp. 138–160. LNCS 4111, Springer (2005). https://doi.org/10.1007/11804192_7

29. Dangl, M., Löwe, S., Wendler, P.: CPACHECKER with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS. pp. 423–425. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_34

30. Darke, P., Prabhu, S., Chimdyalwar, B., Chauhan, A., Kumar, S., Chowdhury, A.B., Venkatesh, R., Datar, A., Medicherla, R.K.: Veriabs: Verification by abstraction and test generation (competition contribution). In: Proc. TACAS. pp. 457–462. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_32

31. D'Silva, V., Kröning, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. on CAD of Integrated Circuits and Systems **27**(7), 1165–1178 (2008). https://doi.org/10.1109/TCAD.2008.923410

32. Ferles, K., Wüstholz, V., Christakis, M., Dillig, I.: Failure-directed program trimming. In: Proc. ESEC/FSE. pp. 174–185. ACM (2017). https://doi.org/10.1145/3106237.3106249

33. Fischer, J., Jhala, R., Majumdar, R.: Joining data flow with predicates. In: Proc. FSE. pp. 227–236. ACM (2005). https://doi.org/10.1145/1081706.1081742

34. Gadelha, M.Y.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: Esbmc v6.0: Verifying C programs using k-induction and invariant inference (competition contribution). In: Proc. TACAS (3). pp. 209–213. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_15
35. Ge, X., Taneja, K., Xie, T., Tillmann, N.: DyTa: Dynamic symbolic execution guided with static verification results. In: Proc. ICSE. pp. 992–994. ACM (2011). https://doi.org/10.1145/1985793.1985971
36. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: Unleashing the power of alternation. In: Proc. POPL. pp. 43–56. ACM (2010). https://doi.org/10.1145/1706299.1706307
37. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: A new algorithm for property checking. In: Proc. FSE. pp. 117–127. ACM (2006). https://doi.org/10.1145/1181775.1181790
38. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: Proc. PLDI. pp. 375–385. ACM (2009). https://doi.org/10.1145/1542476.1542518
39. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Proc. CAV, pp. 526–538. LNCS 2404, Springer (2002). https://doi.org/10.1007/3-540-45657-0_45
40. Holley, L.H., Rosen, B.K.: Qualified data-flow problems. In: Proc. POPL. pp. 68–82. ACM (1980). https://doi.org/10.1145/567446.567454
41. Jakobs, M.C., Wehrheim, H.: Certification for configurable program analysis. In: Proc. SPIN. pp. 30–39. ACM (2014). https://doi.org/10.1145/2632362.2632372
42. Jakobs, M.C., Wehrheim, H.: Programs from proofs: A framework for the safe execution of untrusted software. ACM Trans. Program. Lang. Syst. **39**(2), 7:1–7:56 (2017). https://doi.org/10.1145/3014427
43. Jalote, P., Vangala, V., Singh, T., Jain, P.: Program partitioning: A framework for combining static and dynamic analysis. In: Proc. WODA. pp. 11–16. ACM (2006). https://doi.org/10.1145/1138912.1138916
44. Li, K., Reichenbach, C., Csallner, C., Smaragdakis, Y.: Residual investigation: predictive and precise bug detection. In: Proc. ISSTA. pp. 298–308. ACM (2012). https://doi.org/10.1145/2338965.2336789
45. Necula, G.C.: Proof-carrying code. In: Proc. POPL. pp. 106–119. ACM (1997). https://doi.org/10.1145/263699.263712
46. Necula, G.C., McPeak, S., Weimer, W.: CCured: Type-safe retrofitting of legacy code. In: Proc. POPL. pp. 128–139. ACM (2002). https://doi.org/10.1145/503272.503286
47. Post, H., Sinz, C., Kaiser, A., Gorges, T.: Reducing false positives by combining abstract interpretation and bounded model checking. In: Proc. ASE. pp. 188–197. IEEE (2008). https://doi.org/10.1109/ASE.2008.29
48. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop-invariant generation using splitter predicates. In: Proc. CAV. pp. 703–719. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_57
49. Sherman, E., Dwyer, M.B.: Structurally defined conditional data-flow static analysis. In: Proc. TACAS (2). pp. 249–265. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_15
50. Steffen, B.: Property-oriented expansion. In: Proc. SAS. pp. 22–41. LNCS 1145, Springer (1996). https://doi.org/10.1007/3-540-61739-6_31
51. Thakur, A.V., Govindarajan, R.: Comprehensive path-sensitive data-flow analysis. In: Proc. CGO. pp. 55–63. ACM (2008). https://doi.org/10.1145/1356058.1356066

52. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. In: Proc. POPL. pp. 291–299. ACM (1985). https://doi.org/10.1145/318593.318659
53. Yin, X., Knight, J.C., Weimer, W.: Exploiting refactoring in formal verification. In: Proc. DSN. pp. 53–62. IEEE (2009). https://doi.org/10.1109/DSN.2009.5270355