Decomposing Software Verification using Distributed Summary Synthesis

DIRK BEYER, LMU Munich, Germany
MATTHIAS KETTL, LMU Munich, Germany
THOMAS LEMBERGER, LMU Munich, Germany

There are many approaches for automated software verification, but they are either imprecise, do not scale well to large systems, or do not sufficiently leverage parallelization. This hinders the integration of software model checking into the development process (continuous integration). We propose an approach to decompose one large verification task into multiple smaller, connected verification tasks, based on blocks in the program control flow. For each block, summaries (block contracts) are computed — based on independent, distributed, continuous refinement by communication between the blocks. The approach iteratively synthesizes preconditions to assume at the block entry (computed from postconditions received from block predecessors, i.e., which program states reach this block) and violation conditions to check at the block exit (computed from violation conditions received from block successors, i.e., which program states lead to a specification violation). This separation of concerns leads to an architecture in which all blocks can be analyzed in parallel, as independent verification problems. Whenever new information (as a postcondition or violation condition) is available from other blocks, the verification can decide to restart with this new information. We realize our approach on the basis of configurable program analysis and implement it for the verification of C programs in the widely used verifier CPAchecker. A large experimental evaluation shows the potential of our new approach: The distribution of the workload to several processing units works well, and there is a significant reduction of the response time when using multiple processing units. There are even cases in which the new approach beats the highly-tuned, existing single-threaded predicate abstraction.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; *Formal methods*; • **Computing methodologies** → *Parallel algorithms*; • **Theory of computation** → *Verification by model checking*; *Program reasoning*; • **General and reference** → *Evaluation*.

Additional Key Words and Phrases: Formal Verification, Program Analysis, Software Model Checking, Block Summaries, Contract Synthesis, Decomposition Strategies, Parallelization, Multi-process Model Checking

ACM Reference Format:
Dirk Beyer, Matthias Kettl, and Thomas Lemberger. 2024. Decomposing Software Verification using Distributed Summary Synthesis. *Proc. ACM Softw. Eng.* 1, FSE, Article 59 (July 2024), 23 pages. https://doi.org/10.1145/3660766

1 INTRODUCTION

Despite recent advances [1] in automated software verification and integration into industrial development processes [2–7], the response time of tools for formal software verification does not scale. Comparing the CPU time with the wall time of the results from the International Competition on Software Verification (SV-COMP [1]), it is visible that none of the 52 verification

Authors' Contact Information: Dirk Beyer, Institute for Informatics, LMU Munich, Munich, Germany, dirk.beyer@sosy.ifi.lmu.de; Matthias Kettl, Institute for Informatics, LMU Munich, Munich, Germany, matthias.kettl@sosy.ifi.lmu.de; Thomas Lemberger, Institute for Informatics, LMU Munich, Munich, Germany, thomas.lemberger@sosy.ifi.lmu.de.

ACM 2994-970X/2024/7-ART59
https://doi.org/10.1145/3660766

tools uses significant parallelization of the workload onto the available hardware. This makes formal verification unsuitable for continuous integration [6]. Compositional verification [8] tries to mitigate this: the verification task is divided into multiple, dependent parts. Program parts are analyzed separately, which allows to distribute the verification task onto multiple workers; this, in theory, allows to scale the response time required for verification according to the number of program parts and workers. But there are two issues with this approach: First, program parts are—so far—mostly divided using procedure boundaries. This restricts the possible number of parallel workers to the number of procedures in the program under analysis; scaling to a large number of workers is not possible. Second, program parts depend on each other; there is only a single program specification that the program under analysis is verified against, and if one program part relies on another program part (e.g., one function calls another function), analysis can only continue when the dependent program part is fully analyzed. This makes workers wait on each other, and restricts the amount of actual concurrent work.

Our new approach, *distributed summary synthesis*, solves both issues: Distributed summary synthesis divides the program under analysis into separate blocks. Each block is considered its own verification task, with a given precondition to assume, a block summary (postcondition), and a violation condition to check in addition to the original program specification. All preconditions, block summaries, and violation conditions are continuously refined through program analysis and information from predecessor and successor blocks. Preconditions are refined top-down: If a block's analysis generates a proof with regards to its violation condition and original program specification, the corresponding postcondition at the block exit is propagated to all successor blocks. The violation conditions are refined bottom-up: If an analysis reaches a program state that violates the block's violation condition or the original program specification, it constructs a condition for the block entry that leads to a violation in this block, and propagates it to all predecessor blocks as new violation condition. If a violation condition is propagated up to the program entry, a violation is found that cannot be refuted anymore, and the program is considered unsafe. If all blocks produce a proof, no more violations will be found and the program is considered safe.

We base our formulation of distributed summary synthesis on configurable program analysis [9]. Configurable program analysis is a configurable framework that allows to combine different program analyses and abstract domains. In the scope of this work, we focus on predicate abstraction [10] with counterexample-guided abstraction refinement (CEGAR) [11–13].

Example. The program in Fig. 1 is safe since variables `x` and `y` have the same initial value and are incremented synchronously in the body of the while loop. Therefore, it is impossible for `x` and `y` to have different values at the assert in line 8.

Decomposition. We represent programs as control-flow automata (CFA) (Fig. 2). First, distributed summary synthesis decomposes the CFA into *blocks*: coherent subgraphs with exactly one entry- and one exit-location. Based on their entry- and exit-locations, blocks are connected with each other in a *block graph*. Fig. 3 shows a possible decomposition of Fig. 2. Table 1 shows one possible run of distributed summary synthesis on Fig. 3. For simplicity, we assume in our example that all block-analysis iterations are synchronized. The columns A, B, and C refer to analyses on the respective blocks in Fig. 3.

Iteration 0. All local analyses start with initial states R_{start_0}. The concrete value of R_{start_0} depends on the abstract domain that is used for the analysis and the current block. We use predicate abstraction [10] and start at the block entries, with no initial information about the program states (e.g., initial state $\{pc = l_1\}$ for block A). Initially, the violation conditions are empty, and the target states T_0 for analysis are the states at location l_e (that is reached when the assert fails).

```
1  int main() {
2    int x = 0;
3    int y = 0;
4    while (n()) {
5      x++;
6      y++;
7    }
8    assert(x == y);
9  }
```

Fig. 1. Program

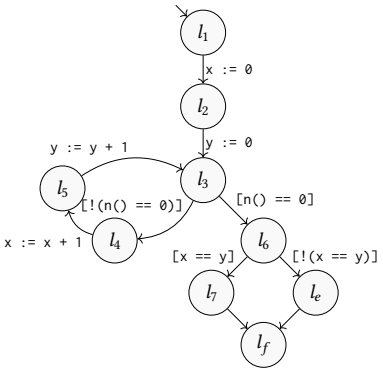

Fig. 2. Control-flow automaton of Fig. 1

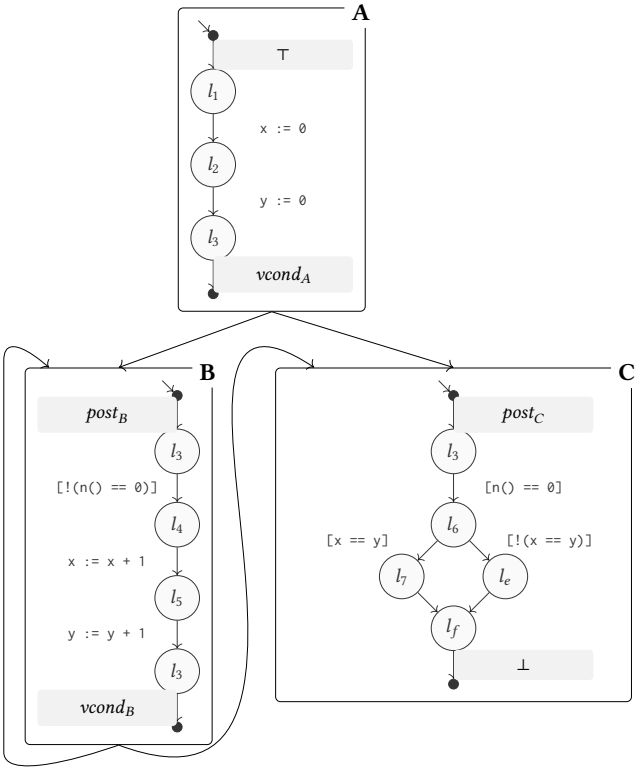

Fig. 3. A valid decomposition of Fig. 2

The analyses on blocks A and B reach no target states (proof) and compute the trivial summary $pc = l_3$ at their block exits and broadcast these summaries as their postconditions to all blocks. Subscript M denotes that the communicated information is packed in a message. The analysis on block C reaches the target state $pc = l_e$ (violation). From that, the analysis computes the violation condition: If, at block entry l_3, condition $x \neq y$ holds, then the program violates the specification. This violation condition is broadcast to all blocks.

Iteration 1. Because blocks A and B are predecessors of block C, they update their target states (T_1) with the information of the violation condition (message from C). The analysis of block A does not reach any state in T_1 and computes the summary (postcondition) $pc = l_3 \wedge x = y$ at its block exit and broadcasts it to the other blocks. Block B reaches target states (violation) and broadcasts a violation condition. Because block C did not receive new information, it is idle in I_1.

Iteration 2. Because blocks B and C are successors of A, they update their initial reached sets R_{start_2} with the new postcondition from A. Block B has two predecessors: block A and itself. Distributed summary synthesis uses Tarjan's algorithm [14] to identify cyclic dependencies (like block B depends on itself). Thus, R_{start_2} for B is set to the postcondition communicated by A. Block C has two non-cyclic dependencies: block A and block B. The postcondition $pc = l_3$ from block B in I_0 is joined with the postcondition $pc = l_3 \wedge x = y$ that is communicated from block A in I_1. The result is the least upper bound $pc = l_3$. The target states T_2 of blocks A and B are updated with the violation condition communicated by B.

The analysis of block A does not reach any target state in T_2 and computes the same summary as before. Block B reaches no target state anymore with its updated initial reached set R_{start_2}, and

Table 1. Distributed summary synthesis on Fig. 3

	A	**B**	**C**
R_{start_0}	$\{pc = l_1\}$	$\{pc = l_3\}$	$\{pc = l_3\}$
T_0	$\{pc = l_e\}$	$\{pc = l_e\}$	$\{pc = l_e\}$
I_0	**proof** $\Rightarrow$ broadcast $(\tau_{post}, \text{A}, \{pc = l_3\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{post}, \text{B}, \{pc = l_3\}_M)$	**violation** $\Rightarrow$ broadcast $(\tau_{vcond}, \text{C}, \{pc = l_3 \wedge x \neq y\}_M)$
R_{start_1}	$\{pc = l_1\}$	$\{pc = l_3\}$	$\{pc = l_3\}$
T_1	$T_0 \cup \{pc = l_3 \wedge x \neq y\}$	$T_0 \cup \{pc = l_3 \wedge x \neq y\}$	T_0
I_1	**proof** $\Rightarrow$ broadcast $(\tau_{post}, \text{A}, \{pc = l_3 \wedge x = y\}_M)$	**violation** $\Rightarrow$ broadcast $(\tau_{vcond}, \text{B},$ $\{pc = l_3 \wedge x + 1 = x'$ $\wedge\, y + 1 = y' \wedge x' \neq y'\}_M)$	*idle because no change*
R_{start_2}	$\{pc = l_1\}$	$\{pc = l_3 \wedge x = y\}$	$\{pc = l_3\}$
T_2	$T_1 \cup \{pc = l_3 \wedge x + 1 = x'$ $\wedge\, y + 1 = y' \wedge x' \neq y'\}$	$T_1 \cup \{pc = l_3 \wedge x + 1 = x'$ $\wedge\, y + 1 = y' \wedge x' \neq y'\}$	T_1
I_2	**proof** $\Rightarrow$ broadcast $(\tau_{post}, \text{A}, \{pc = l_3 \wedge x = y\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{post}, \text{B}, \{pc = l_3 \wedge x = y\}_M)$	*idle because no change*
R_{start_3}	$\{pc = l_1\}$	$\{pc = l_3 \wedge x = y\}$	$\{pc = l_3 \wedge x = y\}$
T_3	T_2	T_2	T_2
I_3	*idle because no change*	*idle because no change*	**proof** $\Rightarrow$ broadcast $(\tau_{post}, \text{C}, \{pc = l_f\})$
		Fixpoint reached, program safe.	

computes also the summary $pc = l_3 \wedge x = y$. Block C is still idle because there is no change in its conditions.

Iteration 3. The new postcondition from block B makes C update its initial reached set R_{start_3} to $pc = l_3 \wedge x = y$. In the last iteration, the analysis on blocks A and B have no new information and are idle. Block C reaches no target state anymore (proof). Now the last broadcast of each block is a proof and a fixed point is reached. The overall verification verdict is that the program is safe.

In summary, we can see that each block was analyzed separately, without any knowledge about the internals of the other blocks or their analysis. The communication is done via two kinds of messages, postconditions and violation conditions.

Contribution. We provide the following contributions:

- We define a stateless, concurrent framework for distributed software verification, called distributed summary synthesis, and implement it for C programs in the formal-verification tool CPAchecker [15].
- We show the benefits of distributed summary synthesis compared to the existing state of the art in a thorough empirical study. For this, we use the largest available benchmark set for the verification of C programs, sv-benchmarks.
- All our implementation and data are available open source.

Related Work. Distributed summary synthesis is inspired by adjustable-block encoding (ABE) [16, 17], block-abstraction memoization (BAM) [18–20], and the SMT-based function summaries of

HiFrog [21]. Adjustable-block encoding (ABE) [16, 17] splits a program into variable-sized blocks that have a predecessor-successor relationship, as well as a nesting hierarchy. Block-abstraction memoization [18–20] introduces a cache for block summaries and unifies the concepts of function and loop summaries. Blocks with a (transitive) predecessor or successor relationship are analyzed sequentially in a forward program analysis, but blocks that are in no predecessor or successor relationship can be analyzed concurrently [19]. The blocks of distributed summary synthesis also have a predecessor-successor relationship, but we decide for a simpler, flat structure. In addition, we do not wait for predecessor blocks to be analyzed, but immediately start computing a summary for each block (that may later be refined). This leads to weaker dependencies between blocks and enables a stronger parallelization. HiFrog [21, 22] splits a program into its individual functions and analyzes each function independently. The summary per function is computed with BMC, Craig interpolation [23, 24], and different SMT theories: summaries are first computed with less-precise, but cheaper integer theories, and on demand, the summaries are recomputed with more-precise bitvector theories. Similar to HiFrog, distributed summary synthesis refines whenever new violation conditions arise. In contrast to HiFrog, distributed summary synthesis can encode summaries on a more fine-grained level than functions: A program can be divided into blocks of arbitrary sizes and can be analyzed with different verification techniques.

Bi-abduction [3, 25] allows compositional program analysis by inferring necessary preconditions for each statement handled (including function calls). Multiple techniques [3, 18, 21, 26–33] use function summaries for interprocedural program analysis. Within our analysis, we use predicate abstraction [10, 34, 35] with counterexample-guided abstraction refinement (CEGAR) [11–13] and Craig interpolation [23, 24]. Other SMT-based approaches to model checking [36–39] are also possible. *DSS* allows the use of any abstract domain to compute summaries.

Portfolio approaches [40] run multiple verification techniques in parallel on the same verification task. This can happen without information exchange between the individual techniques [41–44], or with exchange (for example to 'feed' an invariant checker) [45]. Some approaches [18, 46–50] parallelize the state-space exploration by dividing the program-state space vertically into independent subspaces that are explored separately. Multi-threading is also used by some verification backends [51, 52]. Coverity [50, 53] includes a parallelized, worker-based static analysis. Unlike distributed summary synthesis, Coverity does not refine individual sub-tasks on-demand, but runs five separate, parallelized phases on code blocks to compute pre- and post-conditions.

2 BACKGROUND

For our presentation we only consider intraprocedural, imperative programs with two types of operations: variable assignments and control-flow assumptions. The programs only use arithmetics over bit-vector arithmetics. Our technique can also be used for interprocedural analysis, thanks to its modular nature.

Program Representation. A variable assignment $x := exp$ assigns program variable x the value of expression exp. A control-flow assumption $[p]$ allows the control-flow to pass only if boolean expression p is true. The (infinite) set Ops represents all possible program operations. The special program variable pc is the program counter. It represents the current location in the program. We represent programs as *control-flow automata* (CFA). A CFA $P = (L, l_1, G)$ is a directed graph with program locations L (nodes), program entry $l_1 \in L$ (entry node) and control-flow edges $G : L \times Ops \times L$. A control-flow edge (l, op, l') represents that the control flows from l to l' by setting $pc = l'$ and evaluating op. Initially, $pc = l_1$.

A program path $p = \langle (l_1, op_1, l_2), (l_2, op_2, l_3) \dots (l_{n-1}, op_{n-1}, l_n) \rangle$ is a sequence of connected control-flow edges in the CFA. The set $Paths$ consists of all supposable program paths.

Block-Adjustment Operator. The operator $\texttt{blk} : G \to \mathbb{B}$ (inspired by the *block-adjustment operator* [16]) maps each control-flow edge (l, op, l') of a CFA P to *true* or *false*. If $\texttt{blk}((l, op, l')) = \mathit{true}$, then (l, op, l') is the end of the current block and each outgoing edge $(l', \cdot, \cdot) \in P$ is the beginning of a new block. If $\texttt{blk}((l, op, l')) = \mathit{false}$, then (l, op, l') is part of the current block, only. The concrete definition of blk is not restricted; two trivial definitions are $\texttt{blk}^{sbe}$ and $\texttt{blk}^{false}$. Operator $\texttt{blk}^{sbe}$ always returns true; i.e., every block consists of a single program operation. Operator $\texttt{blk}^{false}$ always returns false; i.e., there is only a single block, and it represents the full program. Block-adjustment operator $\texttt{blk}^{linear}$ produces blocks of linear, non-branching control-flow edges. It returns true for CFA edge (l, op, l') in two cases: First, if the target node l' has two or more incoming edges. This creates a new block if control flows join. Second, if the target node l' has two or more outgoing edges. This creates new blocks if the control flow splits. Formally,
$\texttt{blk}^{linear}((l, op, l')) \;=\; \left|\{(\cdot, \cdot, l') \in G\}\right| > 1 \;\vee\; \left|\{(l', \cdot, \cdot) \in G\}\right| > 1.$

Program Properties. Our approach aims at reachability properties. A reachability property $\varphi = \Box pc \neq l_e$ represents the property that the program never reaches *target location* l_e. Each program state with $pc = l_e$ is a target state.[1]

Configurable Program Analysis. An abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of the set C of all possible concrete program states, the semi-lattice $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$, and a concretization function $[\![\cdot]\!] : E \to 2^C$. The lattice elements E of $\mathcal{E}$ are used as abstract program states. The relation $\sqsubseteq$ is a partial order over E. The join $e \sqcup e'$ of two abstract states is the least upper bound of e and e' in $\mathcal{E}$. The symbol $\top$ denotes the join over E. Each abstract program state represents a set of concrete program states. The concretization function $[\![\cdot]\!]$ maps each abstract state to the set of concrete program states that it represents. We extend $[\![\cdot]\!]$ to sets of abstract states: $[\![S]\!] = \bigcup_{e \in S} [\![e]\!]$ for $S \subseteq E$.

A configurable program analysis (CPA) [54] $\mathbb{D} = (D, \Pi, \rightsquigarrow, \texttt{merge}, \texttt{stop}, \texttt{prec})$ consists of abstract domain D with set E of abstract states, the set of precisions Π, transfer relation $\rightsquigarrow \subseteq E \times G \times (E \times \Pi)$, merge operator $\texttt{merge} : E \times E \times \Pi \to E$, stop operator $\texttt{stop} : E \times 2^E \times \Pi \to \mathbb{B}$, and precision adjustment operator $\texttt{prec} : E \times \Pi \times 2^{(E \times \Pi)} \to E \times \Pi$.

The abstract domain D defines the possible abstract states E, their relation to each other, and their relation to the concrete program states. The set of precisions Π defines the level of abstraction. The transfer relation $\rightsquigarrow \subseteq E \times G \times (E \times \Pi)$ relates an abstract state $e \in E$ to possible successor states $e' \in E$ with precision $\pi \in \Pi$ when evaluating CFA edge $g \in G$, which we denote short as $e \overset{g}{\rightsquigarrow} (e', \pi)$. We write $e \overset{g_1}{\rightsquigarrow} e_1 \overset{g_2}{\rightsquigarrow} \ldots e_n$ to abbreviate $e \overset{g_1}{\rightsquigarrow} (e_1, \pi), \ldots, e_{n-1} \overset{g_n}{\rightsquigarrow} (e_n, \pi)$. The merge operator $\texttt{merge}(e, e', \pi)$ combines the information of abstract states e and e', and produces an abstract state that is equal to or more abstract than e'. The stop operator $\texttt{stop}(e, R, \pi)$ returns *true* if abstract state e is already covered by the set $R \subseteq E$ of known abstract states. It returns *false* otherwise. The precision adjustment operator $\texttt{prec}(e, \pi, R)$ abstracts e to a new abstract state using precision π and the set R of known abstract states.

Reachability Analysis. A reachability analysis $\mathcal{A}(\mathbb{D}, R_0, W_0, T) = (\texttt{reached}, \texttt{waitlist})$ is an algorithm that receives a CPA $\mathbb{D}$, an initial reached set $R_0 \subseteq (E \times \Pi)$, a waitlist $W_0 \subseteq R_0$, and a set $T \subseteq E$ of target states. It uses CPA $\mathbb{D}$ to compute the set $\texttt{reached} \subseteq (E \times \Pi)$ of reachable states (accompanied by the used analysis precision). It assumes that all states in R_0 are reachable and starts computation with the abstract states in W_0. If a target state is computed as reachable, $\mathcal{A}$ may stop the analysis early. In this case, the returned set waitlist contains the *frontier states* that were already computed as reachable, but not considered for further computations yet. If all reachable states are computed, the returned waitlist is empty.

[1] For the sake of simplicity, we only consider reachability of a single target location in our presentation. Any reachability property can be reduced to this through program transformation or monitors. In our evaluation, we use benchmark tasks with multiple potential target locations.

Algorithm 1 $CPA(\mathbb{D}, R_0, W_0, T)$ [55], adapted

Input: CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \texttt{merge}, \texttt{stop}, \texttt{prec})$,
set $R_0 \subseteq (E \times \Pi)$ of abstract states with precision,
set $W_0 \subseteq R_0$ of frontier abstract states with precision,
set $T \subseteq E$ of target states,
where E denotes the set of elements of the semi-lattice of D.
Output: set `reached` of reachable and set `waitlist` of frontier abstract states with precision

```
1:  reached := R_0
2:  waitlist := W_0
3:  while waitlist ≠ ∅ do
4:    pop (e, π) from waitlist
5:    for each e' with e⇝(e', π) do
6:      (ê, π̂) = prec(e', π, reached)
7:      if ⟦ê⟧ ∩ ⟦T⟧ ≠ ∅ then
8:        return reached ∪ {(ê, π̂)}, waitlist
9:      for each (e'', π'') ∈ reached do
10:       e_new := merge(e', e'', π̂)
11:       if e_new ≠ e'' then
12:         waitlist := waitlist ∪ {(e_new, π̂)} \ {(e'', π'')}
13:         reached := reached ∪ {e_new} \ {(e'', π'')}
14:     if ¬stop(e', {e | (e, ·) ∈ reached}, π̂) then
15:       waitlist := waitlist ∪ {(ê, π̂)}
16:       reached := reached ∪ {(ê, π̂)}
17: return reached, waitlist
```

CPA Algorithm. The CPA algorithm [54] $CPA(\mathbb{D}, R_0, W_0, T)$ (Alg. 1) is a reachability analysis. It first initializes the set `reached` of reachable states and the set `waitlist` of states to consider with the corresponding input values (lines 1–2). Then, while there are still states to consider (*frontier states* `waitlist`), an abstract state and the precision (e, π) are popped (line 4) from the waitlist. Each successor e' of e is considered. The algorithm first applies `prec` to e', π, and `reached`. It obtains a new abstract state $\widehat{e}$ and precision $\widehat{\pi}$ (line 6). In lines 7 and 8, the algorithm checks whether $\widehat{e}$ has a common concrete state with set T of target states. If so, a target state is found and the algorithm returns the set `reached` (with the target state) and the current `waitlist`. Otherwise, the algorithm iterates over all pairs of reached states and precisions (e'', π''). It tries to merge the successor state e' with each already-reached state e''. If a merge succeeds (line 12), `waitlist` and `reached` are updated accordingly. Afterwards, the algorithm checks with operator `stop` whether e' itself should be added to `waitlist` and `reached` (line 14). If it should, the precision-adjusted elements $\widehat{e}$ and $\widehat{\pi}$ are added. The algorithm then continues with the next state in `waitlist` (line 3). If all states are explored, it returns the set `reached` and the then-empty `waitlist`.

Counterexample. The abstract counterexample $\texttt{cex} : E \times 2^E \to \mathit{Paths}$ finds, for an abstract state e and set $R \subseteq E$ of reached abstract states (with $e \in R$), a program path $\langle g_1, \ldots, g_n \rangle \in \mathit{Paths}$ so that $e_0 \overset{g_1}{\rightsquigarrow} \ldots \overset{g_n}{\rightsquigarrow} e$ with $e_0, \ldots, e_{n-1} \in R$. The function $\omega : \mathit{Paths} \times E \to 2^E$ finds, for a finite program path $\langle g_1, \ldots, g_n \rangle$ and an abstract state e, all possible abstract states e' so that $e' \overset{g_1}{\rightsquigarrow} \ldots \overset{g_n}{\rightsquigarrow} e$ is feasible.

Counterexample-guided Abstraction Refinement. Algorithm 2 shows counterexample-guided abstraction refinement (CEGAR) [11–13]. This is a reachability analysis that iteratively refines the precision of the analysis based on found counterexamples. Algorithm 2 starts with the abstract states and precision of the initial reached set R_{in} and waitlist W_{in} (lines 1–2). It then uses the

Algorithm 2 *CEGAR*$(\mathbb{D}, R_0, W_0, T)$ [55], adapted

Input: CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \texttt{merge}, \texttt{stop}, \texttt{prec})$,
set $R_0 \subseteq (E \times \Pi)$ of abstract states with precision,
set $W_0 \subseteq R_0$ of frontier abstract states with precision,
set $T \subseteq E$ of target states,
where E denotes the set of abstract states.
Output: set `reached` of reachable and set `waitlist` of frontier abstract states, with precision

```
1:  R_in := R_0
2:  W_in := W_0
3:  while true do
4:    reached, waitlist := CPA(D, R_in, W_in, T)
5:    if waitlist = ∅ then
6:      return reached, waitlist
7:    σ = extractErrorPath(reached)
8:    if isFeasible(σ) then
9:      return reached, waitlist
10:   π' := refine(σ)
11:   R_in := {(e, π ∪ π') | (e, π) ∈ R_in}
12:   W_in := {(e, π ∪ π') | (e, π) ∈ W_in}
```

CPA algorithm to compute the set `reached` of reachable states and the set `waitlist` of frontier states (line 4). If `waitlist` is empty, CPA did not compute any target state; the algorithm returns `reached` and the empty `waitlist` (line 6). If `waitlist` is not empty, a target state was computed; Alg. 2 then extracts a counterexample σ from `reached` and checks whether it is actually feasible. If σ is feasible, the algorithm found a reachable target state and returns `reached` and `waitlist`. If σ is not feasible, the precision of the analysis was too coarse. The algorithm refines the precision based on the infeasible counterexample (line 10) and adds that new precision to the initial reached set and waitlist. It then restarts by running the CPA algorithm with the updated precision (line 4).

Function $\texttt{extractErrorPath} : 2^{(E \times \Pi)} \to \mathit{Paths}$ extracts the error path from the reached set. Function $\texttt{isFeasible} : \mathit{Paths} \to \mathbb{B}$ checks whether the error path is actually feasible. Function $\texttt{refine} : \mathit{Paths} \to \Pi$ takes an infeasible error path and computes a precision that excludes it.

Predicate Abstraction. The predicate analysis CPA [35, 54] $\mathbb{P} = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \texttt{merge}, \texttt{stop}, \texttt{prec}_{\mathbb{P}})$ can use boolean predicate abstraction [17, 35, 56] over a precision $\pi \in \Pi_{\mathbb{P}}$ [55], which is a finite set of predicates (that defines the level of abstraction). The abstract domain $D_{\mathbb{P}}$ is defined using formulas as abstract states. Roughly speaking, the transfer relation $\rightsquigarrow_{\mathbb{P}}$ contains the transfer $\phi \overset{g}{\rightsquigarrow}_{\mathbb{P}} (\phi', \pi)$ if ϕ' is the (satisfiable) strongest postcondition $SP(\phi, op)$, for edge $g = (l, op, l')$. At certain program locations (function entries, exits, loop heads), the precision-adjustment operator `prec` abstracts ϕ' to its boolean predicate abstraction over predicates from π. The merge operator does not combine elements, i.e., $\texttt{merge}(e, e', \pi) = e'$. The $\texttt{stop}(e, R, \pi)$ considers for each state $e' \in R$ separately whether it covers e. For more details, we refer to the literature [35]. To derive the precision π on demand, we use CEGAR [11–13] with lazy abstraction [34, 57], adjustable-block encoding [16, 35], and Craig interpolation [23, 24]. The initial precision π_0 is normally the empty set.

Actor Model. The actor model [58] is a model for distributed computation. Each worker in a distributed system is an *actor* that communicates only through broadcasting messages to all actors in the system, including itself. Actors decide whether and how to handle a received message. An example with four actors is shown in Fig. 4. The boxes represent the actors, the arrows indicate their communication channels.

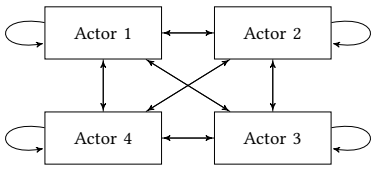

Fig. 4. Actor model with four actors; each actor broadcasts new messages to all actors, including itself

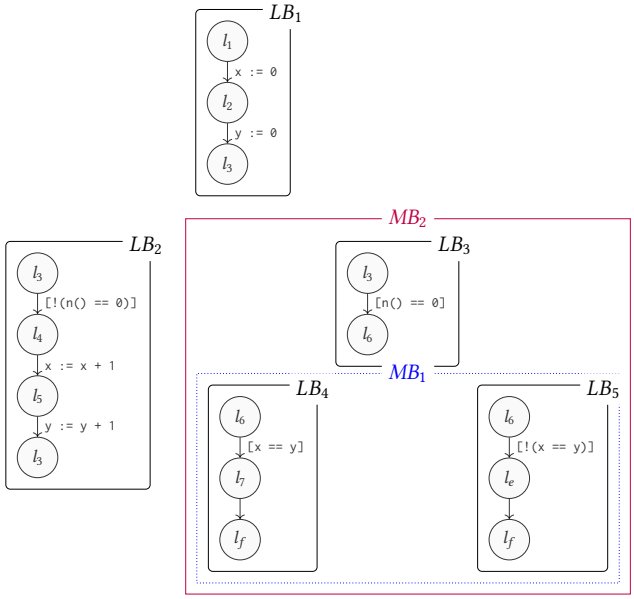

Fig. 5. Decomposition of Fig. 2 with $\texttt{blk}^{linear}$ (blocks LB_i), with a succinct horizontal block merge (block MB_1) and vertical block merge (block MB_2), which results in Fig. 3

3 DISTRIBUTED SUMMARY SYNTHESIS

Decomposition. Given a CFA $P = (L, l_1, G)$, a block b is a weakly connected subgraph $b = (L_b, l_{entry}, l_{exit}, G_b)$ with nodes $L_b \subseteq L$, edges $G_b \subseteq G$, entry node l_{entry} (no predecessor of l_{entry} in L_b), and exit node l_{exit} (no successor of l_{exit} in L_b). There is one exemption to this rule: if a block covers a full loop iteration, then l_{exit} and l_{entry} are both the node at the loop head (see LB_2 in Fig. 5). For each node l in a block, there is a path from l_{entry} to l_{exit} that goes through l.

We decompose P into a *block graph* $\mathcal{B} = (B, G_{\mathcal{B}})$ with a set B of blocks and the directed edges $G_{\mathcal{B}} \subseteq B \times B$ between the blocks. Cycles in the graph are possible.

A *valid decomposition* of $P = (L, l_1, G)$ is a block graph such that for each node $l \in L$ one of the following holds: (a) node l occurs in only one single block or (b) node l connects one block b with all its successor blocks: it (exclusively) is the exit node of b and the entry node of all successor blocks of b. We use the block-adjustment operator $\texttt{blk}$ to flexibly decompose P. Our decomposition traverses the edges of a given CFA and marks the edges for which $\texttt{blk}$ returns true. This divides the CFA in the desired blocks. For technical reasons, our implementation in CPAchecker adds one virtual node (small filled circles in Fig. 3) at the beginning and end of each block, with one nop edge that represents the precondition of the block, and one nop edge that represents the violation condition of the block. Figure 5 shows the (valid) decomposition of Fig. 2 with $\texttt{blk}^{linear}$.

Block Merging. To optimize the number of blocks and their size, we add a strategy for merging blocks both horizontally and vertically until they converge against a given target number of blocks.

We can merge horizontally if two blocks share the same entry and exit location. A horizontal merge of block $b = (L_b, l_{entry}, l_{exit}, G_b)$ and block $b' = (L_{b'}, l_{entry}, l_{exit}, G_{b'})$ results in a new block $b'' = (L_b \cup L_{b'}, l_{entry}, l_{exit}, G_b \cup G_{b'})$ with both blocks' locations and edges, and the common entry and common exit location. In Fig. 5, we can merge LB_4 and LB_5 into MB_1.

We can merge two blocks vertically if the exit location of the first block is the entry location of the second block. Additionally, the first block must be the only block in the block graph with this exit location, and the second block must be the only block in the block graph with this entry location. A vertical merge of block $b = (L_b, l_{entry}, l_{exit}, G_b)$ and block $b' = (L_{b'}, l'_{entry}, l'_{exit}, G_{b'})$ with $l_{exit} = l'_{entry}$ results in a new block $b'' = (L_b \cup L_{b'}, l_{entry}, l'_{exit}, G_b \cup G_{b'})$ with both blocks' locations and edges, the entry location of b, and the exit location of b'.

In Fig. 5 it is not possible to merge blocks LB_3 and LB_4 because LB_4 shares entry location l_6 with LB_5. However, we can merge $LB_3 = (L_{LB_3}, l_3, l_6, G_{LB_3})$ and $MB_1 = (L_{MB_1}, l_6, l_f, G_{MB_1})$ vertically to $MB_2 = (L_{LB_3} \cup L_{MB_1}, l_3, l_f, G_{LB_3} \cup G_{MB_1})$. Our merge strategy alternates between the horizontal and vertical merge until we cannot merge any more blocks or we reach the target number of blocks. The final result of merging the blocks in Fig. 5 horizontally into MB_1 and then vertically into MB_2 is the block graph shown in Fig. 3. In the worst-case, the block-merging operation requires $O(n^2)$ steps, where n is the number of blocks in the block graph. We need to check the conditions for horizontal and vertical merge for every pair of blocks and restart $n - 1$ times if all blocks can be merged into one final block.

Messages. Messages $M \subseteq T \times B \times C$ are three-tuples that consist of a type, a block identifier, and a condition. The type $\tau \in T = \{\tau_{post}, \tau_{vcond}\}$ indicates what kind of condition a message contains. Messages with type τ_{post} transport the postcondition of a block (as a set of abstract states). Messages with type τ_{vcond} transport violation conditions (as a set of target-reaching states). We indicate packed objects by adding the subscript M. For example, $\{e\}_M$ is the set $\{e\}$ packed in a message.

Distributed CPA. We introduce a *distributed* CPA $\mathcal{D} = (\mathbb{D}, \texttt{packPost}, \texttt{packVcond}, \texttt{unpackPost}, \texttt{unpackVcond})$ that defines how to pack and unpack the abstract states E of a CPA $\mathbb{D}$ into and from messages. Operator $\texttt{packPost} : 2^E \times B \to 2^M$ packs a set of abstract states (for a block $b \in B$) into a set of postcondition messages; $\texttt{unpackPost} : 2^M \times B \to 2^E$ unpacks a set of postcondition messages into abstract states; $\texttt{packVcond} : 2^E \times 2^E \times B \to 2^M$ packs found target states into violation-condition messages; and $\texttt{unpackVcond} : 2^M \times B \to 2^E$ unpacks target-reaching states from a set of violation-condition messages.

Distributed Summary Synthesis Algorithm. Algorithm 3 shows the DSS algorithm. It receives the block b to run on, the set E_0^b of initial states, the specification φ, a reachability analysis $\mathcal{A}$, and a distributed CPA $\mathcal{D}$. Algorithm 3 first initializes the program specification's target states T_φ and message lists for received postconditions (`post`) and violation conditions (`vcond`) (line 1–line 3). For each block $b' \in B$, the lists `post` and `vcond` hold the most recently received postcondition and violation condition from block b', respectively. Initially, the postconditions of all predecessors are set to the initial states E_0^b and all violation conditions are set to the empty set. The algorithm then enters a loop (line 4) and waits for a message to arrive (`nextMessage()`). Once a message $m = (\tau, b'_m, \cdot)$ arrives, it is integrated into the corresponding list: If $\tau = \tau_{post}$, line 7 removes the previous message of b'_m from `post` and appends m. Analogously, if $\tau = \tau_{vcond}$, line 9 updates `vcond`.

Line 10 unpacks the postconditions to their set of abstract states, and adds the initial precision π_0 to each state. Line 11 unpacks the violation conditions to the set T of target states for this block (including the original target states T_φ). Then, the algorithm runs the reachability analysis $\mathcal{A}$ on block b with the CPA $\mathbb{D}$, the set R_{start} of initial states *and* states in the waitlist, and the set T of target states. This computes the set R of states that are reachable in block b from R_{start}. Because R includes both abstract states and the used precision, line 13 extracts the set E_R of abstract states. From this, line 14 extracts the set V of reached target states. If V is not empty, the algorithm broadcasts the violation conditions for the block's predecessors (line 16). If V is empty, the algorithm uses E_R to construct and broadcast a stronger postcondition for the block's successors.

Algorithm 3 DSS$(b, E_0^b, \varphi, \mathcal{A}, \mathcal{D})$

Input: Block b, initial states $E_0^b \subseteq E$, specification φ, reachability analysis $\mathcal{A}$, and distributed CPA $\mathcal{D} = (\mathbb{D}, \texttt{packPost}, \texttt{packVcond}, \texttt{unpackPost}, \texttt{unpackVcond})$, where E denotes the set of abstract states and $\pi_0 = \emptyset$ is an initial precision for $\mathbb{D}$.

1: $T_\varphi := \{e \in E \mid e \not\models \varphi\}$
2: $\texttt{post} := [(\tau_{post}, b', E_{0\,M}^b) \mid b' \in B]$
3: $\texttt{vcond} := [(\tau_{vcond}, b', \emptyset_M) \mid b' \in B]$
4: **while** true **do**
5: $\quad m := \texttt{nextMessage}()$
6: $\quad$ **if** $m = (\tau_{post}, b'_m, \cdot)$ **then**
7: $\quad\quad \texttt{post} := [(\tau_{post}, b', \cdot) \in \texttt{post} \mid b' \neq b'_m] \circ [m]$
8: $\quad$ **if** $m = (\tau_{vcond}, b'_m, \cdot)$ **then**
9: $\quad\quad \texttt{vcond} := [(\tau_{vcond}, b', \cdot) \in \texttt{vcond} \mid b' \neq b'_m] \circ [m]$
10: $\quad R_{start} := \{(e, \pi_0) \mid e \in \texttt{unpackPost}(\texttt{post}, b)\}$
11: $\quad T := \texttt{unpackVcond}(\texttt{vcond}, b) \cup T_\varphi$
12: $\quad R, \cdot := \mathcal{A}_b(\mathbb{D}, R_{start}, R_{start}, T)$
13: $\quad E_R := \{e \mid (e, \cdot) \in R\}$
14: $\quad V := \{e \in E_R \mid [\![e]\!] \cap [\![T]\!] \neq \emptyset\}$
15: $\quad$ **if** $V \neq \emptyset$ **then**
16: $\quad\quad$ **broadcast** $\texttt{packVcond}(V, E_R, b)$
17: $\quad$ **else**
18: $\quad\quad$ **broadcast** $\texttt{packPost}(\{e \in E_R \mid e \text{ located at } l_{exit}\}, b)$

Soundness Criteria. The operators packPost and unpackPost are sound, if for each pair $b_0, b_1 \in B$ of blocks, where b_0 is a predecessor of b_1, the unpacked abstract states represent all concrete states that were included in the originally packed set A of abstract states (i.e., no concrete states are lost):

$$\forall A \subseteq E : [\![A]\!] \subseteq [\![\texttt{unpackPost}(\texttt{packPost}(A, b_0), b_1)]\!]$$

The operators packVcond and unpackVcond are sound if, for each pair $b_0, b_1 \in B$ of blocks, where b_0 is a predecessor of b_1, for a set of computed reachable states E_R, and for a set $V \subseteq E_R$ of target states reached from the block entry of b_1, a block analysis that starts with the target-reaching states $R_V = \{(e, \pi_0) \mid e \in \texttt{unpackVcond}(\texttt{packVcond}(V, E_R, b_1), b_0)\}$, with $\pi_0 = \emptyset$, finds all of the previously found (concrete) target states (we use $(x, y)|^1 = x$ to project a pair to the first component):

$$[\![V]\!] \subseteq [\![\mathcal{A}_{b_1}(\mathbb{D}, R_V, R_V, T)|^1]\!].$$

Precision Criteria. The operators packPost and unpackPost are precise, if for each pair $b_0, b_1 \in B$ of blocks, where b_0 is a predecessor of b_1, the unpack only includes concrete states that were included in the originally packed set A of abstract states (i.e., no additional states are hallucinated):

$$\forall A \subseteq E : [\![A]\!] \supseteq [\![\texttt{unpackPost}(\texttt{packPost}(A, b_0), b_1)]\!]$$

The operators packVcond and unpackVcond are precise, if, for each pair $b_0, b_1 \in B$ of blocks, where b_0 is a predecessor of b_1, for a set of computed reachable states E_R, for a set T of potential target states, and for a set $V \subseteq E_R$ of target states reached from the block entry of b_1, a block analysis that starts with the target-reaching states $R_V = \{(e, \pi_0) \mid e \in \texttt{unpackVcond}(\texttt{packVcond}(V, E_R, b_1), b_0)\}$, with $\pi_0 = \emptyset$, only reaches those target states that were reached before (and no additional ones):

$$[\![V]\!] \supseteq [\![\mathcal{A}_{b_1}(\mathbb{D}, R_V, R_V, T)|^1]\!] \cap [\![T]\!].$$

Algorithm 4 $\mathsf{packPost}_A(E_{in}, b)$

Input: Set E_{in} of abstract states, block b
Output: A single message representing the least upper bound of E_{in}

1: **return** $\{(\tau_{post}, b, \{\sqcup E_{in}\}_M)\}$

Algorithm 5 $\mathsf{packVcond}_A(V, E_R, b)$

Input: Set V of reached target states, reached states E_R, block b
Output: A single message representing the violation condition for V

1: $W := \bigcup_{v \in V} \omega(\mathsf{cex}(v, E_R), v)$
2: **return** $\{(\tau_{vcond}, b, W_M)\}$

Algorithm 6 $\mathsf{unpackPost}_A(M_{post}, b)$

Input: List M_{post} of messages, block b
Output: Least upper bound of abstract states

1: $\mathtt{states} := \{\}$
2: **for** $(\tau_{post}, b', A_M) \in M_{post}$ **do**
3: **if** $b' \in \mathsf{predecessors}(b)$ **then**
4: $\mathtt{states} := \mathtt{states} \cup A$
5: **if** $\mathtt{states} := \{\}$ **then**
6: **return** $\{\top\}$
7: **return** $\{\bigsqcup \mathtt{states}\}$

Algorithm 7 $\mathsf{unpackVcond}_A(M_{vcond}, b)$

Input: List M_{vcond} of messages, block b
Output: Set of target states for block b

1: $T := \{\}$
2: **for** $(\tau_{vcond}, b', W_M) \in M_{vcond}$ **do**
3: **if** $b' \in \mathsf{successors}(b)$ **then**
4: $T := T \cup W$
5: **return** T

Termination Conditions. For each block node in the block graph, we run one separate instance of the DSS algorithm. The instances are organized in an actor model and communicate messages to all actors. We reach a fixed point in one of the following two cases: (1) If all instances' last broadcast contains no more violation conditions and all violation conditions have been processed, no more refinements are necessary. In this case, we stop all DSS instances and report that program P is safe. (2) If a block with no predecessor broadcasts a message of type τ_{vcond}. Because the broadcasting block has no predecessor, future refinements will never be able to exclude this violation. In this case, we terminate all DSS instances and report that program P is unsafe.

Context-Aware Instantiation. We formulate a distributed CPA $\mathcal{D}$ that is aware of its predecessor and successor blocks in the block graph: Operator $\mathsf{packPost}_A$ (Alg. 4) joins all abstract states into a single abstract state that is then packed into a postcondition message, and $\mathsf{unpackPost}_A$ (Alg. 6) unpacks abstract states from given messages M_{post} for block b. It only considers messages that describe postconditions sent from a predecessor block of b. If no such message exists, set $\mathtt{states}$ stays empty and $\mathsf{unpackPost}_A$ returns the top element $\top$. If at least one relevant message existed, $\mathsf{unpackPost}_A$ returns the join of the unpacked states.

Operator $\mathsf{packVcond}_A$ (Alg. 5) collects, for the given set V of target states, those abstract states at the block entry of b from which at least one target state $e \in V$ is reachable; this is computed with $\omega(\mathsf{cex}(v, E_R), v)$ and collected in set W. The states W are the target-reaching states for V and packed into a single violation-condition message. Operator $\mathsf{unpackVcond}_A$ (Alg. 7) restores the abstract states from violation-condition messages M_{vcond} by collecting, from successors of b, all sets W of target-reaching states.

Soundness of Context-Aware Instantiation. Operators $\mathsf{packPost}_A$ and $\mathsf{unpackPost}_A$ are sound:

$$\begin{aligned}
&\mathsf{unpackPost}_A(\mathsf{packPost}_A(E_{in}, b_0), b_1) \\
&= \mathsf{unpackPost}_A(\{(\tau_{post}, b_0, \{\textstyle\bigsqcup E_{in}\}_M)\}, b_1) \\
&= \textstyle\bigsqcup\{\bigsqcup E_{in}\} \;=\; \bigsqcup E_{in}
\end{aligned}$$

with $b_0, b_1 \in B$ and b_0 is a predecessor of b_1. By the definition of the join operator follows the soundness, since $\forall e \in E_{in} : e \sqsubseteq \bigsqcup E_{in}$ and $[\![E_{in}]\!] \subseteq [\![\bigsqcup E_{in}]\!]$. Operators $\mathsf{packVcond}_A$ and $\mathsf{unpackVcond}_A$ are sound. Let

$$\begin{aligned} T_{b_0} &= \mathsf{unpackVcond}_A(\mathsf{packVcond}_A(V, E_R, b_1), b_0) \\ &= \mathsf{unpackVcond}_A(\{(\tau_{vcond}, b_1, W_M)\}, b_0) \\ &= \bigcup_{v \in V} \omega(\mathsf{cex}(v, E_R), v) \end{aligned}$$

with $b_0, b_1 \in B$ and b_0 is a predecessor of b_1. We set $R_V = \{(e, \pi_0) \mid e \in T_{b_0}\}$. By construction of the counterexamples, we find for every $v \in V$ a path from an $e_v \in R_V$ to v. Therefore, $V \subseteq \mathcal{A}_{b_1}(\mathbb{D}, R_V, R_V, T)|^1$ as the analysis explores all reachable states from R_V through the respective counterexample. This implies the soundness criterion $[\![V]\!] \subseteq [\![\mathcal{A}_{b_1}(\mathbb{D}, R_V, R_V, T)|^1]\!]$.

Precision of Context-Aware Instantiation. The operators $\mathsf{packPost}_A$ and $\mathsf{unpackPost}_A$ are not precise, but overapproximate the state-space, as the joins in $\mathsf{packPost}_A$ and $\mathsf{unpackPost}_A$ produce the least-upper-bound of the communicated abstract states. The operators $\mathsf{packVcond}_A$ and $\mathsf{unpackVcond}_A$ with $R_V = \{(e, \pi_0) \in \mathsf{unpackVcond}_A(\mathsf{packVcond}_A(V, E_R, b_1), b_0)\}$, $b_0, b_1 \in B$, and b_0 predecessor of b_1, are precise: $[\![V]\!] \supseteq [\![\mathcal{A}_{b_1}(\mathbb{D}, R_V, R_V, T)|^1]\!] \cap [\![T]\!]$.

It is sufficient to show that every concrete state $c_v \in [\![\mathcal{A}_{b_1}(\mathbb{D}, R_V, R_V, T)|^1]\!] \cap [\![T]\!]$ is contained in $[\![V]\!]$. Let us assume that $c_v \in [\![\mathcal{A}_{b_1}(\mathbb{D}, R_V, R_V, T)|^1]\!]$ and $c_v \in [\![T]\!]$. Then, $\mathcal{A}_{b_1}$ detected a violation described in T starting from the unpacked states. Since cex computes exactly the states that lead to the violation, the violation must have been uncovered before. Therefore, $c_v \in [\![V]\!]$.

Postconditions in Cyclic Block Graphs. We identify strongly connected components (SCC) in the *block graph* with Tarjan's algorithm [14]. Since initially post equals E_0^b for all predecessors, and the postconditions of the predecessors in the same SCC depend on the postconditions of blocks in the same SCC, the analysis always considers at least the abstract states in E_0^b. Thus, the postcondition of a block in an SCC cannot get stronger. To avoid this, we apply the following strategy to all blocks in SSCs: Whenever a block is part of an SSC, we prevent all postconditions from predecessors that are also part of the same component from being joined unless they are unequal to E_0^b. This ensures that we unroll loops at least once and the postcondition of predecessors in the same strongly connected component eventually gets stronger. We can only find valid proofs if all postconditions in the strongly connected component reached a fixed point. We reach a fixed point if no block in the SSC computes a new postcondition containing unseen concrete states.

Context-Free Instantiation. *DSS* is flexible and can also use a context-free instantiation, where the distributed CPA $\mathcal{D}$ implements the four operators as follows (with block b, abstract states E, $A \subseteq E$, $V \subseteq E$):

$$\begin{aligned} \mathsf{packPost}_I(A, b) &= \{(\tau_{post}, b, A_M)\} \\ \mathsf{unpackPost}_I(M, \cdot) &= \bigsqcup \{A \mid (\tau_{post}, \cdot, A_M) \in M\} \\ \mathsf{packVcond}_I(V, E_R, b) &= \{(\tau_{vcond}, b, V_M)\} \\ \mathsf{unpackVcond}_I(M, \cdot) &= \bigsqcup \{V \mid (\tau_{vcond}, \cdot, V_M) \in M\} \end{aligned}$$

The unpack operators unpack all received abstract states and violation conditions without any modification. This DSS instantiation leads to a modular analysis similar to Infer [3], where each code block (in Infer: function) is analyzed separately. The block graph for Infer consists of one block node for each function in the input program with no edges. Nested function calls need to be over-approximated accordingly by setting affected global and local variables to $\top$. If all functions are safe, the approach finds a proof. A violation condition from one function suffices to prove the program unsafe. The behavior is sound, but this approach likely report many false alarms.

4 EVALUATION

4.1 Research Questions

We evaluate our approach along the following research questions:

RQ 1: Distribution of Work Load to Processing Units. Is the approach of *distributed summary synthesis* effective in distributing the verification work to different threads?
Evaluation Plan: We compare the CPU-time consumption of *DSS* with its response time, using 1, 2, 4, and 8 processing units.

RQ 2: Reduction of Response Time. Does using more processing units lead to a significant reduction of the response time when using *distributed summary synthesis*?
Evaluation Plan: We compare the response time of *DSS* using 1, 2, 4, and 8 processing units.

RQ 3: Outperform Predicate Abstraction on Some Programs. Is the new approach able to outperform a 15-years highly-tuned approach on appropriate verification tasks?
Evaluation Plan: We compare *DSS* with 8 processing units to a standard single-threaded predicate abstraction. We select a few verification tasks which employ a sufficient number of workers and block size to see whether we can outperform this state-of-the-art algorithm.

RQ 4: Complement State-of-the-Art Tools. Is the new approach already able to complement state-of-the-art approaches?
Evaluation Plan: We compare *DSS* to the state-of-the-art approaches IMC [24, 59] and k-Induction, both implemented in CPAchecker [45].

RQ 5: Parallel Portfolio. How does the new approach perform in a parallel-portfolio approach that aims to optimize response time?
Evaluation Plan: We compare the performance of the parallel-portfolio of predicate abstraction and *DSS* to standalone predicate abstraction in CPAchecker [16].

4.2 Experiment Setup

We executed our experiments with version 3.16 [2] of BenchExec [60] on 167 machines each having a GNU/Linux operating system (x86_64-linux, Ubuntu 22.04 with Linux kernel 5.15), an Intel Xeon E3-1230 v5 CPU with 3.40 GHz and 8 processing units, and 15 GB of RAM. We evaluate our approach on the reach-safety tasks of the sv-benchmark set [3], which is a large collection of diverse verification tasks in the C programming language. This benchmark set is regularly maintained and used by several tool competitions (e.g., [61, 62]).

In our evaluation, we focus on the safe verification tasks of category *SoftwareSystems*. For finding bugs, the performance largely depends on the traversal order, while for proving correctness, all paths have to be considered. It is important to mention that we do not aim at improving an individual program analysis with our approach, but to distribute the workload over many threads. If the underlying analysis, for example, unrolls a loop several times and does not terminate, our approach will still face the same problem. Since our approach only has limited support for arrays and pointers, we exclude 64 tasks where disabled pointer aliasing causes wrong results for predicate abstraction. A list of the tasks can be found in our artifact [63]. The *SoftwareSystems* category contains tasks from real-world programs and is therefore suitable for our evaluation. This selection defines a benchmark set of 2 485 tasks.

We implemented distributed summary synthesis in revision 46372 of the open-source software-verification framework CPAchecker [4] [15]. CPAchecker is implemented in Java with many components readily available. For decomposition of the CFA, we use $\texttt{blk}^{linear}$ with block merging. In all

[2] https://github.com/sosy-lab/benchexec/releases/tag/3.16
[3] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/commit/3d65c76d
[4] https://svn.sosy-lab.org/software/cpachecker/branches/dss-fse@46372

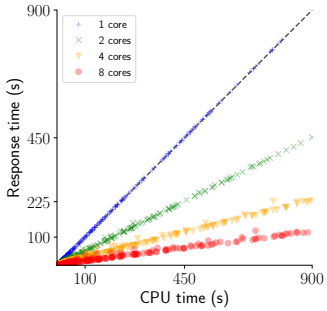

Fig. 6. CPU time (x-axis) and response time (y-axis) for *DSS* with 1, 2, 4, and 8 cores

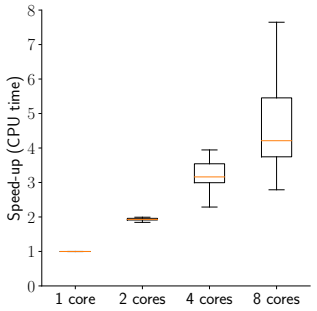

Fig. 7. Speed-up of *DSS* with 1, 2, 4, and 8 cores; the speed-up is the ratio of the CPU to response time

experiments, we instantiate distributed summary synthesis (*DSS*, Alg. 3) with CEGAR (Alg. 2) as reachability analysis and the context-aware distributed CPA for predicate abstraction. The experimental results are available as reproduction package on Zenodo [63].

4.3 Experimental Results

RQ 1: Distribution of Work Load to Processing Units. The scatter plot in Fig. 6 shows, for each task, the required CPU time (in seconds) of *DSS* on the x-axis and the actual response time (in seconds) on the y-axis. For a fixed value on the x-axis, a lower value on the y-axis shows that more work is done effectively in parallel. A data point at 450 s and 225 s indicates that the verification requires 450 s of work (CPU time), but already responds after 225 s due to effective parallelization. For our baseline we run *DSS* on 1 core (+) and observe that the response time equals the required CPU time. With 2 (×), 4 (▽), and 8 (○) cores, *DSS* is able to deliver the same verdict two, three, and four times faster than the baseline, respectively.

The parallelization possible with *DSS* becomes even more visible when considering the box plot in Fig. 7. The use of 2 cores speeds up the analysis response time by the factor 2. The use of 8 cores speeds up the analysis response time by the factor 4, on average. This shows that the approach benefits from more processing units. Amdahl's Law [64] states that n cores allow a maximum speed-up of only below factor n. Currently, the average, measurable speed-up of *DSS* with eight cores is 4, so half of the theoretical upper bound.

> Yes, the new approach effectively distributes the workload to different processing units.

RQ 2: Reduction of Response Time. To show the reduction of the response time, we measure the response time for different numbers of processing units. The box plot in Fig. 8 shows the speed-up of the response time (in seconds) of *DSS* with 2, 4, and 8 cores compared to the response time with 1 core. All tasks benefit from the parallelization as the speed-up compared to *DSS* with 1 core is greater than one. Figure 8 shows that the use of 2 cores already speeds up the response time by about 1.26 for 75 % of the tasks (lower border of the box). For 50 % of the tasks, the response time is sped up by a factor of 1.34 or more. For 5 % of the tasks (upper whisker of the box), the response time is sped up by a factor of 1.70. The theoretical speed-up that is not achievable (Amdahl's Law) is 2. This is excelled by some tasks because the parallelization changes the behavior of the analysis. SMT queries may change due to the parallel computation of conditions, which can lead to a different, more efficient behavior in the SMT solver.

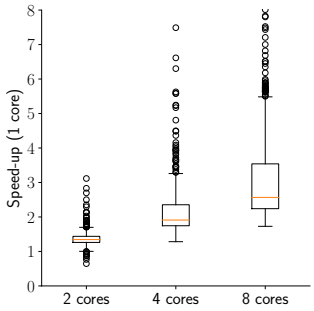

Fig. 8. Speed-up (y-axis) of response time with 2, 4, 8 cores compared to *DSS* with only 1 core; the response time decreases significantly when adding more cores

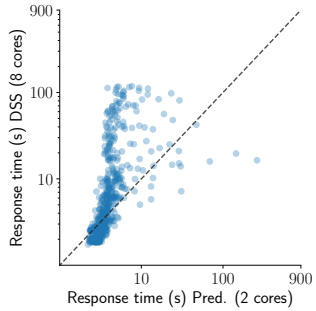

Fig. 9. Comparison of consumed response time of single-threaded predicate abstraction (x-axis) with 2 cores to distributed summary synthesis with 8 cores (y-axis)

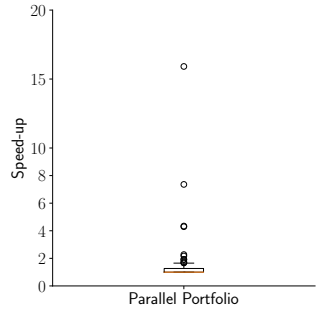

Fig. 10. Speed-up of the parallel portfolio of *DSS* and predicate abstraction compared to standalone predicate abstraction

Table 2. Five tasks where distributed summary synthesis (*DSS*) proves tasks significantly faster than predicate abstraction ($\mathbb{P}$), on 8 cores, with regards to response time; the columns list CPU time (CPU), response time (RT), number of threads spawned by *DSS*, and average number of CFA edges in the blocks of the *DSS* decomposition

Task	$\text{CPU}_{\mathbb{P}}(s)$	$\text{CPU}_{DSS}(s)$	$\text{RT}_{\mathbb{P}}(s)$	$\mathbf{RT}_{\boldsymbol{DSS}}(s)$	# threads	∅ block size
leds–leds-regulator...	44.8	33.2	30.8	7.18	92	12.40
rtc–rtc-ds1553.ko-l...	49.0	64.6	30.3	14.0	164	14.98
rtc–rtc-stk17ta8.ko...	46.7	67.9	28.9	15.1	162	15.70
watchdog–it8712f_w...	86.8	50.3	69.0	15.9	216	7.91
ldv-commit-tester/m0...	50.1	103	28.8	21.0	230	7.00

The response-time speed up compared to 1 core further improves for 4 cores and 8 cores: For 4 cores, the median speed-up is 1.91. For 8 cores, the median speed-up is 2.57. In both cases, the speed-up for the upper quartile increases further.

> Yes, the new approach significantly reduces the response time, and increasing resources to 2, 4, or 8 processing units leads to a significant reduction of the response time.

RQ 3: Comparison to Predicate Abstraction. Figure 9 compares the response time of predicate analysis (x-axis) and the response time of distributed summary synthesis (y-axis) on tasks solved correctly by both approaches. Every data point below the line indicates that this verification task benefits from the distribution provided by *DSS*. The award-winning predicate abstraction is well-known and was highly-tuned since 15 years. We compare with predicate abstraction using two processing units instead of one, because CPAchecker gets some parallelization for free from the garbage collector although the actual analysis algorithm is single-threaded, which makes it more challenging for DSS to beat it. Generally, the highly-tuned predicate abstraction is still faster, as we experience an overhead for the (un)packing and processing of irrelevant messages. Predicate analysis outperforms *DSS* for the vast majority of tasks below the 10 s mark. However, there is a considerable number of tasks that already profit from the distribution. Harder tasks are generally solved faster with *DSS*, if *DSS* finds a proof. As mentioned before, our goal here is to develop a strategy allowing a scalable approach to software model checking. We believe that the potential of the approach is high as we have many opportunities for improvement as discussed later.

Table 3. Verification results of IMC, k-Induction, predicate analysis, and distributed summary synthesis (*DSS*)

Technique	Correctly solved	Timeout	Out of memory	Error
k-Induction	**985**	450	28	1 022
Predicate	860	**696**	14	915
IMC	707	24	15	**1 739**
DSS	592	264	**1 117**	512

Table 2 lists five hand-picked tasks from the benchmark set where distributed summary synthesis outperformes single-threaded predicate analysis. Subscript $\mathbb{P}$ shows the data for predicate analysis, subscript *DSS* shows data for distributed summary synthesis.

In these cases, distributed summary synthesis decreases the response time (RT) significantly. We observe that the number of workers (# threads) and the number of control-flow edges in the blocks (∅ block size) can vary. Many blocks with less edges seem to work as well as fewer blocks with more edges. One of the biggest challenges in the future will be the fine-tuning of the decompositions. Currently, we focused on inexpensive decompositions because the decomposition also produces overhead. However, more expensive strategies might be beneficial in the long run. Some tasks are decomposed into up to 751 blocks, but the solved task with the highest number of blocks is only decomposed into 476 blocks. Hence, equally many threads are spawned.

'Unstructured' control-flow edges such as `goto`, `continue`, and `break` hinder us from merging blocks, potentially resulting in many small blocks.

> Yes, there are some verification tasks for which the new approach even outperforms single-threaded predicate analysis.

RQ 4: Complement State-of-the-Art Tools. The focus of this work is to use existing components. *DSS* tries to complement existing techniques to make them scalable by re-using and not re-inventing them. An encoding of the specification in the violation condition is sufficient to make other existing techniques applicable to *DSS*.

The comparison in Table 3 of our approach (last row) shows that established techniques like IMC or k-Induction are also capable of solving the tasks that *DSS* solves. *DSS* solves 7 tasks that are not solved by k-Induction, 14 not solved by predicate analysis, and 61 not solved by IMC. Taking a deeper look into the 14 tasks that *DSS* solves but predicate abstraction does not, we see that in 9 cases, predicate abstraction runs into the time limit, while *DSS* is faster and finishes within the time limit. The remaining 5 cases are due to tool crashes that *DSS* can avoid due to the different analysis order. However, in general, the 'classic' approaches solve significantly more tasks overall.

Occasionally, we encounter exceptions that are not occurring for the plain predicate analyses. Here, we face unsupported features, e.g., `memcopy`, which the standard predicate abstraction does not traverse because the path can already be excluded with the refined precision after the first iterations of CEGAR. Also, some assertions about valid block graphs might be violated, in which case we exit the verification with an exception. We also encounter significantly more *out-of-memory* exceptions due to the concurrent work done: Since we perform multiple analyses in parallel, we also require more memory. Even for small programs, ARGs might grow significantly, and our approach has to (un)pack thousands of abstract states. In the future, we envision a compressed message format and a different communication model to tackle this problem.

For a fair comparison, we disable pointer aliasing for the plain predicate analysis. Given that the verification verdicts always equal the verification verdicts of the predicate analysis, we assume that our implementation works properly.

No, the new approach is not yet good enough to complement the state-of-the-art verification tools with regards to effectiveness.

RQ 5: Parallel Portfolio. To examine the potential of *DSS* to reduce the response time of verification, we create a parallel portfolio of predicate abstraction and *DSS*. In the parallel portfolio, we run both predicate abstraction and *DSS* in parallel. As soon as one of the techniques produces a verification result, the analysis ends and this result is used. This portfolio combines the strengths of both techniques and mitigates their weaknesses. Figure 10 shows the speed-up of the response time of parallel portfolio, compared to standalone predicate abstraction. Because the parallel portfolio always uses the fastest produced result, it is never slower than predicate abstraction. In 25 % of the cases, the parallel portfolio is at least 27 % faster. In the best case, the portfolio is 16 times faster.

A parallel portfolio with *DSS* improves the response time in our experiments up to factor 16.

Decomposition. Our experiments show that *DSS* benefits from a block graph where program loops are not contained within a single block, but represented by a cycle in the block graph. This ensures that *DSS* generates a block summary for the loop body. We also observe that it is helpful to create blocks that cover one or more full linear sequences of statements; for example a full loop body, a complete if-branch, or both a full if- and else-branch up to their join. This reduces the complexity of block summaries because they do not need to talk about variables with a scope that is fully contained within the block. Our horizontal and vertical merge strategies try to achieve the above. Figure 3 shows how our decomposition represents loops in the block graph (block **B**).

Threats to Validity. Our benchmark set is limited to 2 485 safe tasks and might be biased towards a selection of features of the C programming language. However, sv-benchmarks is the largest and most diverse benchmark set for verification of C programs to date. The software-systems category includes excerpts of real-world software systems as well as specialized algorithms.

We showcase distributed summary synthesis on the example of predicate abstraction. The approach is generic and independent of the abstract domain, but our experimental findings may not generalize to all other abstract domains. Our implementation may contain bugs, but we did not observe any wrong results in our experiments.

We serialize messages for information exchange and broadcast messages to all actors. Experimental results may change if a different communication model is used, but we expect results to only improve with more sophisticated communication models.

A different scheduling of messages might lead to different results and, in the worst-case, cause more unrollings of loops and therefore more messages.

Limitations. Distributed summary synthesis currently has the following limitations:

Context Sensitivity. We set operator $\texttt{packPost}_A$ to run the analysis with a join of all postconditions. This is an overapproximation that loses precision. Due to this, the presented version of distributed summary synthesis is context-insensitive. To still handle cycles in the block graph (due to recursion or loops) successfully, we deploy an optimization based on Tarjan's algorithm (see Sect. 3). Future work could define a more precise operator `packPost` that handles postconditions separately.

Resource Consumption. We spend resources on synthesizing summaries for unreachable program states. In sequential analysis, unreachable program states are usually not computed because the analysis reasons about unreachability sequentially (or iteratively sequentially in the case of CEGAR). However, distributed summary synthesis analyzes blocks in parallel and may unnecessarily compute a program state space that is only proven unreachable later.

```
1  int main() {
2    int a = 0;
3    int b = 0;
4    if (a) hard1(a);
5    if (b) hard2(b);
6  }
```

Fig. 11. Program with two unreachable, hard-to-verify function calls

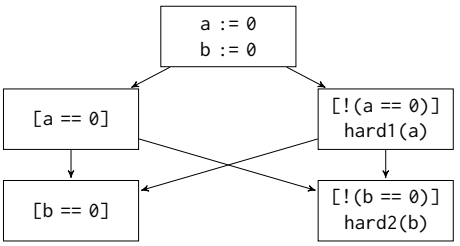

Fig. 12. Poor decomposition of Fig. 11

```
1  int main() {
2    int a = n();
3    if (a) hard1(a);
4    else hard2(a);
5  }
```

Fig. 13. Program with two reachable, hard-to-verify function calls

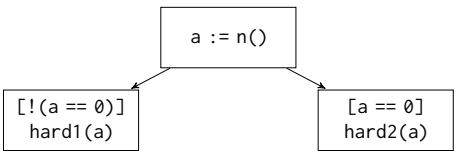

Fig. 14. Good decomposition of Fig. 13

Choice of Decomposition. The issue of unnecessary computations worsens with an unfitting decomposition and can be mitigated with a good decomposition. Figure 11 shows a program with multiple calls to hard-to-verify functions (`hard1`, `hard2`). Because of the initial variable assignment, none of these calls are reachable. A sequential analysis may realize this and will never try to analyze the unreachable functions. But an unfitting decomposition of the verification problem (for example as in Fig. 12) may lead to individual workers for functions `hard1` and `hard2`, so that these are analyzed eagerly. This causes expensive, unnecessary computations. Contrary, if hard blocks are computed in parallel as seen for the program in Fig. 13 in the decomposed version in Fig. 14, the benefits of *DSS* become visible. *DSS* simultaneously works on the proof of both hard-to-verify functions hard1 and hard2. In the case that both are safe, *DSS* can immediately conclude that the whole program is safe. If a block reports a violation, *DSS* only has to check whether the value of *a* can be different from or equal to 0, respectively. *DSS* is perfect for independent blocks like this. In general, *DSS* performs better on tasks with multiple potential target locations, as it can prove their (in)feasibility in parallel. In our experiments, the decomposition produces a negligible overhead. It takes, on average, less than 0.2 % of the total CPU time of *DSS*.

Communication Model. We choose the actor model [58] to have a simple method of communication between blocks. New conditions are sent to every other actor. In case a condition is not meant for its block, i.e., it is not a successor or predecessor, the message is discarded. But for this, every message has to be unpacked first. This causes overhead. In our experiments, packing messages produces a negligible overhead. It takes, on average, less than 1.2 % of the total CPU time of *DSS*. But in contrast, unpacking messages takes, on average, 60 % of the total CPU time of *DSS*. We expect that this overhead significantly reduces through a more sophisticated communication model. The discrepancy of consumed time for (un)packing arises from the fact that all blocks need to unpack a message that was only packed once by the sender.

Scheduling Model. Currently, all blocks are analyzed concurrently and the scheduling of the different blocks is done by the operating system. Therefore, the needed CPU- and response-time of a verification run may differ between runs. The presented experimental data supports the claim that the impact is not observable, but orchestrated scheduling techniques may avoid unnecessary work and reduce the number of sent messages.

Verification Witnesses. Our approach can not yet write verification witnesses [65]. For violation witnesses, we would need to restore the path to the violation. For correctness witnesses, we can use the loop summaries as invariants.

Opportunities. The relevance of *DSS* can be underlined with the possibilities for future research:

Incremental Verification. We envision the incremental verification of larger programs that are actively under development (for example in a continuous-integration pipeline). For this we can exploit the full potential of our approach as we can store the current states of all blocks until the program is patched. Now, we only need to re-analyze the blocks that are affected by the changes. In large software projects this might save a significant bit of work while preserving previously computed information. With the help of the (de)serialization, we can even restore the state of our approach at any time by replaying the logged messages.

Contract Synthesis. DSS constructs block contracts. If the blocks are configured to coincide with functions, then the negated violation condition of a function can serve as its precondition, and the postconditions of each block can be used as such without further changes. Currently, state of the art in interactive verification is to manually derive pre- and postconditions of functions, which might become partially automated using *DSS*.

Verification Techniques. While we evaluated our approach with the abstract domain of predicate abstraction, it is not limited to a particular abstract domain, and other approaches for software model checking can be applied as well. Further, the block contracts can be synthesized by different analysis techniques and verified in isolation, as well as exported for further processing by external tools or the user. This creates the opportunity to integrate theorem provers for contracts.

Strategy Selection. In distributed summary synthesis, the used abstract domains can change per block. This can be exploited by performing strategy selection [66, 67] to use the best-suiting abstract domain for each block.

Decompositions. Other program-decomposition strategies may be explored in the future. Similar to AI-based strategy selection for verification techniques [67], we could use AI for finding suiting decompositions of the program.

5 CONCLUSION

This work presents distributed summary synthesis (*DSS*), an approach for the automatic construction of program contracts and a flexible framework to scale verification through decomposition and distribution. This is achieved by extending configurable program analysis to a distributed setting. *DSS* decomposes a single, large software-verification task into multiple smaller tasks. The parallel and continuously refined synthesis of block summaries (postconditions and violation conditions) reduces the dependencies between blocks. This increases the potential parallelism and allows to distribute the verification work to many processing units. Through this, distributed summary synthesis achieves short response times and enables verification technology for incremental verification and continuous integration [6]. Furthermore, DSS can be used to synthesize code contracts. Our experimental results are promising and show that distributed summary synthesis significantly reduces the response time if multiple processing units are available.

Data-Availability Statement. The data from our experiments are available at Zenodo [63] and on our supplementary webpage: https://www.sosy-lab.org/research/distributed-summary-synthesis/

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY) and 418257054 (Coop).

REFERENCES

[1] D. Beyer. 2024. State of the art in software verification and witness validation: SV-COMP 2024. In *Proc. TACAS (3) (LNCS 14572)*. Springer, 299–329. https://doi.org/10.1007/978-3-031-57256-2_15

[2] T. Ball, V. Levin, and S. K. Rajamani. 2011. A decade of software model checking with Slam. *Commun. ACM* 54, 7 (2011), 68–76. https://doi.org/10.1145/1965724.1965743

[3] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. 2015. Moving fast with software verification. In *Proc. NFM (LNCS 9058)*. Springer, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

[4] 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, 165–176. https://doi.org/10.1007/978-3-642-11486-1_14

[5] B. Cook. 2018. Formal reasoning about the security of Amazon web services. In *Proc. CAV (2) (LNCS 10981)*. Springer, 38–47. https://doi.org/10.1007/978-3-319-96145-3_3

[6] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle. 2021. Code-level model checking in the software development workflow at Amazon Web Services. *Softw. Pract. Exp.* 51, 4 (2021), 772–797. https://doi.org/10.1002/spe.2949

[7] A. Wilson, A. Nötzli, A. Reynolds, B. Cook, C. Tinelli, and C. W. Barrett. 2023. Partitioning strategies for distributed SMT solving. In *Proc. FMCAD*. IEEE, 199–208. https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_28

[8] K. Laster and O. Grumberg. 1998. Modular model checking of software. In *Proc. TACAS (LNCS 1384)*. Springer, 20–35. https://doi.org/10.1007/BFb0054162

[9] 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

[10] 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

[11] 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

[12] T. Ball and S. K. Rajamani. 2000. *Boolean programs: A model and process for software analysis.* Technical Report MSR Tech. Rep. 2000-14. Microsoft Research. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2000-14.pdf.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2000. Counterexample-guided abstraction refinement. In *Proc. CAV (LNCS 1855)*. Springer, 154–169. https://doi.org/10.1007/10722167_15

[14] R. E. Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. https://doi.org/10.1137/0201010

[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*. FMCAD, 189–197. https://ieeexplore.ieee.org/document/5770949.

[17] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. 2009. Software model checking via large-block encoding. In *Proc. FMCAD*. IEEE, 25–32. https://doi.org/10.1109/FMCAD.2009.5351147

[18] D. Wonisch and H. Wehrheim. 2012. Predicate analysis with block-abstraction memoization. In *Proc. ICFEM (LNCS 7635)*. Springer, 332–347. https://doi.org/10.1007/978-3-642-34281-3_24

[19] D. Beyer and K. Friedberger. 2018. Domain-independent multi-threaded software model checking. In *Proc. ASE*. ACM, 634–644. https://doi.org/10.1145/3238147.3238195

[20] D. Beyer and K. Friedberger. 2020. Domain-independent interprocedural program analysis using block-abstraction memoization. In *Proc. ESEC/FSE*. ACM, 50–62. https://doi.org/10.1145/3368089.3409718

[21] L. Alt, S. Asadi, H. Chockler, K. Even-Mendoza, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina. 2017. HiFrog: SMT-based function summarization for software verification. In *Proc. TACAS (LNCS 10206)*. 207–213. https://doi.org/10.1007/978-3-662-54580-5_12

[22] S. Asadi, M. Blicha, G. Fedyukovich, A. E. J. Hyvärinen, K. Even-Mendoza, N. Sharygina, and H. Chockler. 2018. Function summarization modulo theories. In *Proc. LPAR (EPiC, Vol. 57)*. EasyChair, 56–75. https://doi.org/10.29007/d3bt

[23] W. Craig. 1957. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.* 22, 3 (1957), 250–268. https://doi.org/10.2307/2963593

[24] K. L. McMillan. 2003. Interpolation and SAT-based model checking. In *Proc. CAV (LNCS 2725)*. Springer, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

[25] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. 2011. Compositional shape analysis by means of bi-abduction. *J. ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[26] D. Babic and A. J. Hu. 2008. Calysto: Scalable and precise extended static checking. In *Proc. ICSE*. ACM, 211–220. https://doi.org/10.1145/1368088.1368118

[27] T. Ball and S. K. Rajamani. 2000. Bebop: A symbolic model checker for boolean programs. In *Proc. SPIN (LNCS 1885)*. Springer, 113–130. https://doi.org/10.1007/10722468_7

[28] A. Albarghouthi, A. Gurfinkel, and M. Chechik. 2012. Whale: An interpolation-based algorithm for inter-procedural verification. In *Proc. VMCAI (LNCS 7148)*. Springer, 39–55. https://doi.org/10.1007/978-3-642-27940-9_4

[29] T. W. Reps, S. Horwitz, and M. Sagiv. 1995. Precise interprocedural data-flow analysis via graph reachability. In *Proc. POPL*. ACM, 49–61. https://doi.org/10.1145/199448.199462

[30] T. W. Reps. 1997. Program analysis via graph reachability. In *Proc. ILPS*. MIT, 5–19.

[31] T. A. Henzinger, R. Jhala, and R. Majumdar. 2004. Race checking by context inference. In *Proc. PLDI*. ACM, 1–13. https://doi.org/10.1145/996841.996844

[32] B. Stein, B.-Y. E. Chang, and M. Sridharan. 2021. Demanded abstract interpretation. In *Proc. PLDI*. ACM, 282–295. https://doi.org/10.1145/3453483.3454044

[33] M. Heizmann, J. Hoenicke, and A. Podelski. 2010. Nested interpolants. In *Proc. POPL*. ACM, 471–482. https://doi.org/10.1145/1706299.1706353

[34] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. 2007. The software model checker Blast. *Int. J. Softw. Tools Technol. Transfer* 9, 5-6 (2007), 505–525. https://doi.org/10.1007/s10009-007-0044-z

[35] D. Beyer, M. Dangl, and P. Wendler. 2018. A unifying view on SMT-based software verification. *J. Autom. Reasoning* 60, 3 (2018), 299–335. https://doi.org/10.1007/s10817-017-9432-6

[36] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. 1999. Symbolic model checking without BDDs. In *Proc. TACAS (LNCS 1579)*. Springer, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[37] D. Beyer and M. Dangl. 2020. Software verification with PDR: An implementation of the state of the art. In *Proc. TACAS (1) (LNCS 12078)*. Springer, 3–21. https://doi.org/10.1007/978-3-030-45190-5_1

[38] A. F. Donaldson, L. Haller, D. Kröning, and P. Rümmer. 2011. Software verification using k-induction. In *Proc. SAS (LNCS 6887)*. Springer, 351–368. https://doi.org/10.1007/978-3-642-23702-7_26

[39] K. L. McMillan. 2006. Lazy abstraction with interpolants. In *Proc. CAV (LNCS 4144)*. Springer, 123–136. https://doi.org/10.1007/11817963_14

[40] B. A. Huberman, R. M. Lukose, and T. Hogg. 1997. An economics approach to hard computational problems. *Science* 275, 7 (1997), 51–54. https://doi.org/10.1126/science.275.5296.51

[41] D. Baier, D. Beyer, P.-C. Chien, M. Jankola, M. Kettl, N.-Z. Lee, T. Lemberger, M. Lingsch-Rosenfeld, M. Spiessl, H. Wachowitz, and P. Wendler. 2024. CPAchecker 2.3 with strategy selection (competition contribution). In *Proc. TACAS (3) (LNCS 14572)*. Springer, 359–364. https://doi.org/10.1007/978-3-031-57256-2_21

[42] S. Löwe, M. U. Mandrykin, and P. Wendler. 2014. CPAchecker with sequential combination of explicit-value analyses and predicate analyses (competition contribution). In *Proc. TACAS (LNCS 8413)*. Springer, 392–394. https://doi.org/10.1007/978-3-642-54862-8_27

[43] P. Peringer, V. Šoková, and T. Vojnar. 2020. PredatorHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In *Proc. TACAS (2) (LNCS 12079)*. Springer, 408–412. https://doi.org/10.1007/978-3-030-45237-7_30

[44] D. Beyer, S. Kanav, and C. Richter. 2022. Construction of verifier combinations based on off-the-shelf verifiers. In *Proc. FASE*. Springer, 49–70. https://doi.org/10.1007/978-3-030-99429-7_3

[45] 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

[46] G. J. Holzmann. 1997. The Spin model checker. *IEEE Trans. Softw. Eng.* 23, 5 (1997), 279–295. https://doi.org/10.1109/32.588521

[47] J. Barnat, P. Rockai, V. Still, and J. Weiser. 2015. Fast, dynamically-sized concurrent hash table. In *Proc. SPIN (LNCS 9232)*. Springer, 49–65. https://doi.org/10.1007/978-3-319-23404-5_5

[48] G. Yang, R. Qiu, S. Khurshid, C. S. Pasareanu, and J. Wen. 2019. A synergistic approach to improving symbolic execution using test ranges. *Innov. Syst. Softw. Eng.* 15, 3-4 (2019), 325–342. https://doi.org/10.1007/s11334-019-00331-9

[49] Y. Xie and A. Aiken. 2007. Saturn: A scalable framework for error detection using boolean satisfiability. *TOPLAS* 29, 3 (2007), 16. https://doi.org/10.1145/1232420.1232423

[50] S. McPeak, C. H. Gros, and M. K. Ramanathan. 2013. Scalable and incremental software bug detection. In *Proc. ESEC/FSE*. ACM, 554–564. https://doi.org/10.1145/2491411.2501854

[51] S. Blom, J. van de Pol, and M. Weber. 2010. LTSmin: Distributed and symbolic reachability. In *Proc. CAV (LNCS 6174)*. Springer, 354–359. https://doi.org/10.1007/978-3-642-14295-6_31

[52] T. van Dijk. 2016. *Sylvan: Multi-core decision diagrams*. Ph. D. Dissertation. University of Twente, Enschede, Netherlands. http://purl.utwente.nl/publications/100676

[53] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Gros, A. Kamsky, S. McPeak, and D. R. Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75. https://doi.org/10.1145/1646353.1646374
[54] D. Beyer, S. Gulwani, and D. Schmidt. 2018. Combining model checking and data-flow analysis. In *Handbook of Model Checking*. Springer, 493–540. https://doi.org/10.1007/978-3-319-10575-8_16
[55] 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
[56] T. Ball, A. Podelski, and S. K. Rajamani. 2001. Boolean and Cartesian abstraction for model checking C programs. In *Proc. TACAS (LNCS 2031)*. Springer, 268–283. https://doi.org/10.1007/3-540-45319-9_19
[57] 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
[58] C. Hewitt. 2015. Actor model of computation: Scalable robust information systems. *arXiv/CoRR* 1008, 1459 (2015). https://doi.org/10.48550/arXiv.1008.1459
[59] D. Beyer, N.-Z. Lee, and P. Wendler. 2024. Interpolation and SAT-based model checking revisited: Adoption to software verification. *J. Autom. Reasoning* (2024). https://doi.org/10.1007/s10817-024-09702-9 Preprint: https://doi.org/10.48550/arXiv.2208.05046.
[60] D. Beyer, S. Löwe, and P. Wendler. 2019. Reliable benchmarking: Requirements and solutions. *Int. J. Softw. Tools Technol. Transfer* 21, 1 (2019), 1–29. https://doi.org/10.1007/s10009-017-0469-y
[61] D. Beyer. 2023. Competition on software verification and witness validation: SV-COMP 2023. In *Proc. TACAS (2) (LNCS 13994)*. Springer, 495–522. https://doi.org/10.1007/978-3-031-30820-8_29
[62] D. Beyer. 2023. Software testing: 5th comparative evaluation: Test-Comp 2023. In *Proc. FASE (LNCS 13991)*. Springer.
[63] D. Beyer, M. Kettl, and T. Lemberger. 2024. Reproduction package for FSE 2024 article 'Decomposing software verification using distributed summary synthesis'. Zenodo. https://doi.org/10.5281/zenodo.11563223
[64] G. M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS*. ACM, 483–485. https://doi.org/10.1145/1465482.1465560
[65] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig. 2022. Verification witnesses. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 57:1–57:69. https://doi.org/10.1145/3477579
[66] D. Beyer and M. Dangl. 2018. Strategy selection for software verification based on boolean features: A simple but effective approach. In *Proc. ISoLA (LNCS 11245)*. Springer, 144–159. https://doi.org/10.1007/978-3-030-03421-4_11
[67] C. Richter, E. Hüllermeier, M.-C. Jakobs, and H. Wehrheim. 2020. Algorithm selection for software validation based on graph kernels. *Autom. Softw. Eng.* 27, 1 (2020), 153–186. https://doi.org/10.1007/s10515-020-00270-x

Received 2023-09-28; revised 2024-03-06; accepted 2024-04-16