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 are computed — based on independent, distributed, continuous refinement by communication between the blocks. The approach iteratively synthesizes preconditions to assume at the block entry (which program states reach this block) and verification conditions to check at the block exit (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 precondition or verification condition) is available from other blocks, the verification can decide to restart with this new information. We realize our approach as an extension of the configurable-program-analysis algorithm 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**; • **Computing methodologies** → *Parallel algorithms*; • **General and reference** → *Evaluation.*

Additional Key Words and Phrases: Program Analysis, Software Model Checking, Block Summaries, Decomposition Strategies, Parallelization

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 [2–5], 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 tools uses significant parallelization of the workload onto the available hardware. This makes formal verification unsuitable for continuous integration [5]. Compositional verification [6] tries

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

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

```
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

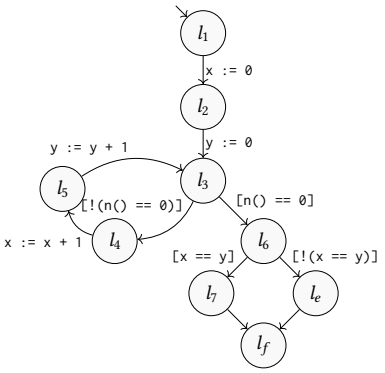

Fig. 2. Control-flow automaton of Fig. 1

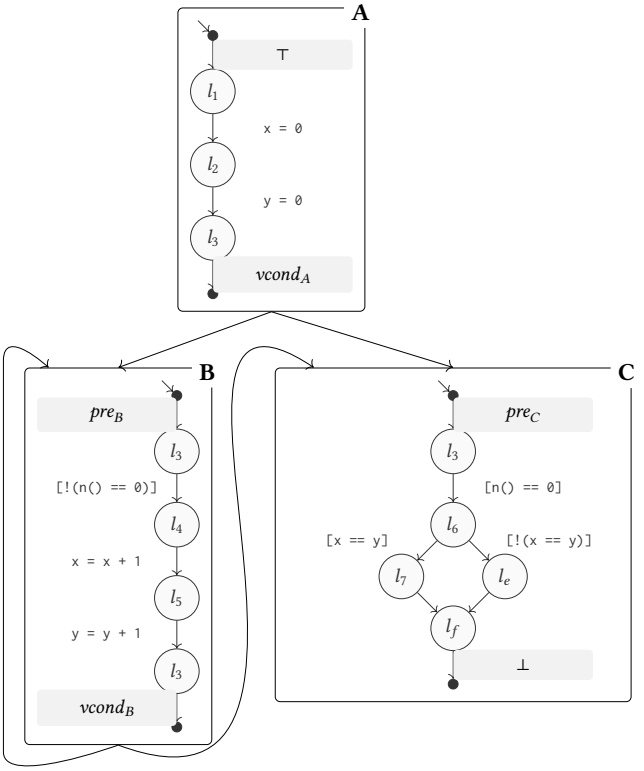

Fig. 3. A valid decomposition of Fig. 2

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, 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 block summary at the block exit is propagated as new preconditions to all successor blocks. The violation conditions are refined bottom-up: If an analysis reaches a program state that violates its violation condition or the original program specification, it abducts a condition for the

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_1	**proof** $\Rightarrow$ broadcast $(\tau_{pre}, \text{A}, \{pc = l_3\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{pre}, \text{B}, \{pc = l_3\}_M)$	**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_2	**proof** $\Rightarrow$ broadcast $(\tau_{pre}, \text{A}, \{pc = l_3 \wedge x = y\}_M)$	**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_3	**proof** $\Rightarrow$ broadcast $(\tau_{pre}, \text{A}, \{pc = l_3 \wedge x = y\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{pre}, \text{B}, \{pc = l_3 \wedge x = y\}_M)$	*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_4	*idle because no change*	*idle because no change*	**proof** $\Rightarrow$ broadcast $(\tau_{pre}, \text{C}, \{pc = l_f\})$
	Fixpoint reached, program safe.		

violation in this block, and propagates it to all predecessor blocks as new violation condition. When a violation condition is propagated from the program entry, a violation is found that cannot be refuted anymore, and the program is considered unsafe. When all blocks produce a proof, no more violations will be found and the program is considered safe.

We define distributed summary synthesis as an extension of the configurable-program-analysis algorithm [7] to a distributed setting, called DCPA. Configurable program analysis is a configurable framework for program analysis that allows to combine different analyses and abstract domains. In the scope of this work, we focus on predicate abstraction [8].

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.

Analysis Setup. Table 1 shows one possible run of distributed summary synthesis on Fig. 3. For simplicity, we assume in our example that all workers' analysis iterations are synchronized. The columns A, B, and C refer to analyses on the respective blocks in Fig. 3. 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 [8] 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 the error location l_e.

Iteration 1. In the first iteration, the analyses on blocks A and B reach no target states and compute the trivial summary $pc = l_3$ at their block exits. The analysis on block C reaches the target state $pc = l_e$. From that, 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. Because blocks A and B are predecessors of block C, they update their target states (T_1) with the information of the violation condition.

Iteration 2. Because block C did not receive new information, it is idle. The next analysis (I_2) on block A does not reach any state in T_1 and computes the summary $pc = l_3 \wedge x = y$ at its block exit. Because blocks B and C are successors of A, they update their initial reached sets R_{start_2} with this new precondition from A. Block B has two successors: block A and itself. Distributed summary synthesis uses Tarjan's algorithm [9] to identify cyclic dependencies like block B has on itself, and sets R_{start_2} for B with the precondition communicated by A. Block C has two non-cyclic dependencies: block A and block B. The precondition $pc = l_3$ that is communicated by block B is joined with the precondition $pc = l_3 \wedge x = y$ that is communicated by block A. 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.

Iteration 3. Block A does not reach any target state in T_2 and computes the same summary as before. With its updated initial reached set R_{start_2}, the analysis on block B reaches no target state anymore and computes the summary $pc = l_3 \wedge x = y$. Block C is still idle because of no change in its precondition or violation condition. This updated summary of block B makes C update its initial reached set R_{start_3} to $pc = l_3 \wedge x = y$.

Iteration 4. In the last iteration, blocks A and B have no new information and are idle, and the analysis on block C reaches no target state anymore (produces a proof). Now the last broadcast of each block is a proof and a fixpoint is reached. Therefore, the overall verification verdict is that the program is safe.

Contribution. We provide the following contributions:

- We define a generic extension of configurable program analysis to a distributed setting, called DCPA. Based on this we define *distributed summary synthesis*, a stateless, concurrent approach to scale automated software verification.
- We implement distributed summary synthesis for C programs in the formal-verification tool CPAchecker [10].
- 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) [11, 12], block-abstraction memoization (BAM) [13, 14], and the SMT-based function summaries of HiFrog [15]. Adjustable-block encoding (ABE) [11, 12] splits a program into variable-sized blocks that have a predecessor-successor relationship, as well as a nesting hierarchy. Block-abstraction memoization [13, 16] introduces a cache for block summaries and generalizes the concept 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 [14]. 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 [15, 17] splits a program into its individual functions and analyzes each function independently. The summary computed per function is computed with BMC, Craig interpolation [18], and different SMT theories: summaries are first computed with less-precise, but cheaper integer theories. 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 more fine-grained than function summaries: A program ca be divided into blocks of arbitrary sizes and can be analyzed with different verification techniques.

Bi-abduction [3, 19] allows compositional program analysis by inferring necessary preconditions for each statement handled (including function calls). Multiple techniques [3, 13, 15, 20–26] use function summaries for interprocedural program analysis. Within our analysis, we use predicate abstraction [8] with counterexample-guided abstraction refinement [27] and Craig interpolation [28]. Other SMT-based approaches to model checking [29–32] are also possible. Many techniques [15, 21, 22, 33] use Craig-interpolation [28] to construct summaries; but they first build a full Boolean program of the function to summarize, and then apply Craig interpolation to the full Boolean representation. Our approach differs from them: it allows to use any SMT-based approach to compute summaries, and Craig interpolation is not used to compute the block summaries, but to derive predicates for predicate abstraction.

Portfolio approaches run multiple verification techniques in parallel on the same verification task. This can happen without information exchange between the individual techniques [15, 34, 35], or with exchange (for example to 'feed' an invariant checker) [36]. Some approaches [13, 37–41] 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 the backends [42, 43]. Coverity [41, 44] 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 presentation, we consider programs written in a simple programming language: we consider intraprocedural, imperative programs with two types of operations: variable assignments and control-flow assumptions. Later we show that our technique can be used for interprocedural analysis without adjustments, thanks to its modular nature.

Program Representation. A variable assignment $x := exp$ assigns program variable x the value of expression exp. We only consider arithmetic expressions over unbound integers. A control-flow assumption $[p]$ only allows the control-flow to pass 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, $pc = l_1$.

Algorithm 1 $CPA(\mathbb{D}, R_0, W_0)$ [45], adapted

Input: CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$,
set $R_0 \subseteq E$ of abstract states,
set $W_0 \subseteq R_0$ of frontier abstract states,
where E denotes the set of elements of the semi-lattice of D.
Output: set reached of reachable abstract states

```
1:  reached := R_0
2:  waitlist := W_0
3:  while waitlist ≠ ∅ do
4:    pop e from waitlist
5:    for each e' with e⇝e' do
6:      for each e'' ∈ reached do
7:        e_new := merge(e', e'')
8:        if e_new ≠ e'' then
9:          waitlist := waitlist ∪ {e_new} \ {e''}
10:         reached := reached ∪ {e_new} \ {e''}
11:     if ¬stop(e', reached) then
12:       waitlist := waitlist ∪ {e'}
13:       reached := reached ∪ {e'}
14: return reached
```

A program path $p = \langle (l, op, l'), (l', op', l'') \dots (l''', op'', l'''') \rangle$ is a sequence of connected control-flow edges in the CFA. Every program path is element of the set *Paths*.

Block-Adjustment Operator. The *block-adjustment operator* [11] $\text{blk}(l, op, l')$ maps each control-flow edge (l, op, l') of a CFA P to $true$ or $false$. If $\text{blk}(l, op, l') = true$, 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 $\text{blk}(l, op, l') = false$, then (l, op, l') is part of the current block, only. The concrete definition of blk is arbitrary; two trivial definitions are blk^{sbe} and blk^{false}. Operator blk^{sbe} always returns true; i.e., every block consists of a single program operation. Operator blk^{false} always returns false; i.e., there is only a single block, and it represents the full program. Block-adjustment operator blk^{linear} produces blocks of linear, non-branching control-flow edges. It returns true for CFA edge (l, op, l') in two cases: First, when the target node l' has two or more incoming edges. This creates a new block when control flows join. Second, when the target node l' has two or more outgoing edges. This creates new blocks when the control flow splits. Formally, $\text{blk}^{linear}(l, op, l') = \left|\{(\cdot, \cdot, l_{in}) \in G \mid l_{in} = l'\}\right| > 1 \vee \left|\{(l_{out}, \cdot, \cdot) \in G \mid l_{out} = l'\}\right| > 1$.

Program Properties. Our approach aims at reachability properties. A reachability property $\varphi = \square pc \neq l_e$ 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 possible, concrete program states, the semi-lattice $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$, and a concretization function $[\![\cdot]\!] : E \to 2^C$. The elements E of $\mathcal{E}$ are used as abstract program states. Relation $\sqsubseteq$ is a partial order over E, with $\forall e \in E : e \sqsubseteq \top$. The join $e \sqcup e'$ of two abstract states is the least upper bound of e and e' in

[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. In our evaluation, we use benchmark tasks with multiple potential target locations.

$\mathcal{E}$. 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 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) [45] $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$ consists of abstract domain D with set E of abstract states, transfer relation $\rightsquigarrow \subseteq E \times G \times E$, merge operator $\text{merge} : (E \times E) \times E$, and stop operator $\text{stop} : (E \times 2^E \times \mathbb{B})$. The abstract domain D defines the possible abstract states E, their relation to each other, and their relation to the concrete program states. The transfer relation $\rightsquigarrow \subseteq E \times G \times E$ relates each abstract state $e \in E$ to its successor states $e' \in E$ when evaluating CFA edge $g \in G$. We write $e \overset{g}{\rightsquigarrow} e'$. The merge $\text{merge}(e, e')$ 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 $\text{stop}(e, R)$ returns true if abstract state e is already covered by the set $R \subseteq E$ of known abstract states. It returns false otherwise.

CPA Algorithm. The CPA algorithm [45] $CPA(\mathbb{D}, R_0, W_0)$ (Alg. 1) receives a CPA $\mathbb{D}$, an initial reached set $R_0 \subseteq E$, and a waitlist $W_0 \subseteq R_0$. It uses CPA $\mathbb{D}$ to compute the set reached $\subseteq E$ of reachable states. It assumes that all states in R_0 are reachable and starts computation with the abstract states in W_0. In detail, Alg. 1 first initializes the reachable states reached 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 e is popped (line 4). Each successor e' of e is considered. The algorithm first tries to merge e' with each already reached abstract state $e'' \in$ reached. If a merge succeeds (line 8), waitlist and reached are updated accordingly. Afterwards, the algorithm checks with stop whether e' itself should be added to waitlist and reached (lines 11–13). It then continues with the next state in waitlist (line 3). When all states are explored, the algorithm returns the set reached of reachable states. Variants [46] of the CPA algorithm do not explore the full set of reachable states, but terminate as soon as a target state is found.

Counterexample. The abstract counterexample $\text{cex} : E \times 2^E \rightarrow \mathit{Paths}$ 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 $\omega : \mathit{Paths} \times E \rightarrow 2^E$ 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.

Predicate Abstraction. The predicate analysis CPA [47] $\mathbb{P} = (D_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \text{merge}, \text{stop})$ implements predicate abstraction [8] with Boolean predicate abstraction [48] over a finite set π of predicates. The abstract domain $D_{\mathbb{P}}$ is defined as Boolean formulas over π. Formulas use static single-assignment form. The transfer relation $\rightsquigarrow_{\mathbb{P}}$ contains the transfer $r \overset{g}{\rightsquigarrow}_{\mathbb{P}} r'$ if for edge $g = (l, op, l')$, the strongest post condition $SP(r, op)$ is satisfiable and r' is the strongest Boolean predicate abstraction of $SP(r, op)$ with predicates of π. The merge operator does not combine elements, $\text{merge}(e, e') = e'$. The $\text{stop}(e, \text{reached})$ separately considers for each state $e' \in$ reached whether it covers e. To derive the predicates π on-line, we use counterexample-guided abstraction refinement [27] and Craig interpolation [28]. To reduce the number of Boolean predicate abstractions, we use adjustable block encoding [11] and only compute abstractions at function entries, exits, and loop heads.

Actor Model. The actor model [49] 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.

3 DISTRIBUTED SUMMARY SYNTHESIS

Decomposition. Given CFA $P = (L, l_1, G)$, a block b is a weakly connected subgraph $b = (L_b, l_{entry}, l_{exit}, G_b) \subseteq P$ with nodes $L_b \subseteq L$, edges $G_b \subseteq G$, entry node l_{entry} (no predecessor

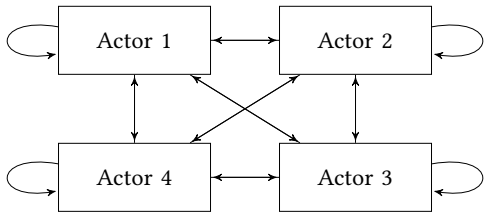

Fig. 4. Actor model with four actors; each actor broadcasts new messages to all actors, including itself

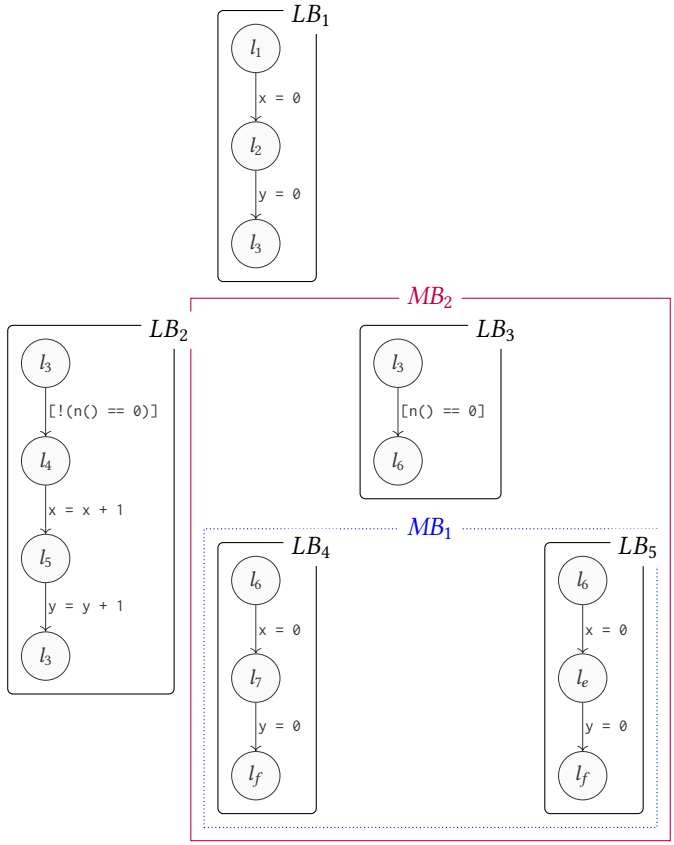

Fig. 5. Decomposition of Fig. 2 with blk^{linear} (blocks LB_i). With a succinct horizontal block merge (block MB_1) and vertical block merge (block MB_2). This results in Fig. 3.

of l_{entry} in L_b), and exit node l_{exit} (no successor of l_{exit} in L_b). There is one exemption from this rule: if a block covers a full loop iteration, then l_{exit} and l_{entry} are at the loop head (see block B in Fig. 3). For each node l in the 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 is a block graph so that for each node $l \in L$ in the CFA one of the following holds: (a) Node l only occurs in a 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 blk to flexibly decompose P. Our decomposition traverses the edges of a given CFA and marks the edges for which blk returns true. This divides the CFA in the desired blocks. This decomposition can be efficiently implemented as depth-first search with a runtime complexity of $O(|L| + |G|)$, where $|L|$ is the number of locations in the CFA and $|G|$ is the number of CFA edges. For technical reasons with CPAchecker, we add one virtual node 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 blk^{linear}.

Block Merging. To keep the number of blocks manageable, 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 $b' = (L_{b'}, l'_{entry}, l'_{exit}, G_{b'})$ 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 alters 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, all blocks of a block graph can be merged in $O(n(n-1))$, 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 are three-tuples $T \times B \times C$. The type $\tau \in T = \{\tau_{pre}, \tau_{vcond}\}$ indicates what kind of content $\zeta \in C$ block $b \in B$ sent. Messages with type τ_{pre} transport a set of abstract states. Messages with type τ_{vcond} transport violation conditions. We indicate packed objects by adding the subscript M. For example, $\zeta = \{e\}_M$ is the packed set $\{e\}$.

Distributed CPA Algorithm. We extend the CPA algorithm to a *distributed CPA* (DCPA) algorithm. The DCPA algorithm receives a CPA $\mathbb{D}$, the block b to run on, the set R_0^b of initial states, the specification φ, and four additional operators: packPre, packVcond, unpackPre, and unpackVcond. Operator packPre : $2^E \times B \to 2^M$ packs a set of abstract states E at block b into a set of messages, operator unpackPre : $2^M \times B \to 2^E$ unpacks a set of messages for block b into abstract states, operator packVcond : $2^E \times B \to 2^M$ packs violation conditions at block b into messages, and operator unpackVcond : $2^M \times B \to 2^E$ unpacks from a set of messages for block b violation conditions.

Algorithm 2 first initializes the program specification's target states T_φ and message lists for received preconditions (pre) and violation conditions (vcond) (line 1–line 3). List pre holds, for each block β, the most recent received precondition of β. List vcond holds, for each block β, the most recent received violation condition of β. Initially, all preconditions are set to the initial states R_0^b and all violation conditions are set to the empty set. The algorithm then enters a loop (line 4)

Algorithm 2 DCPA($\mathbb{D}, b, R_0^b, \varphi$, packPre, packVcond, unpackPre, unpackVcond)

Input: CPA $\mathbb{D}$ with abstract elements E,
block b, initial states $R_0^b \subseteq E$, specification φ,
operators packPre, packVcond, unpackPre, and unpackVcond.

1: $T_\varphi := \{e \mid e \in E \wedge e \not\models \varphi\}$ wrt. a specification φ
2: pre $:= [(\tau_{pre}, \beta, R_{0\,M}^b) \mid \beta \in B]$
3: vcond $:= [(\tau_{vcond}, \beta, \emptyset_M) \mid \beta \in B]$
4: **while** true **do**
5: $m :=$ waitForNextMessage()
6: **if** $m = (\tau_{pre}, \beta_m, \cdot)$ **then**
7: pre $:=$ [$(\tau_{pre}, \beta, \cdot) \in$ pre $\mid \beta \neq \beta_m$] $\circ$ [m]
8: **if** $m = (\tau_{vcond}, \beta_m, \cdot)$ **then**
9: vcond $:=$ [$(\tau_{vcond}, \beta, \cdot) \in$ vcond $\mid \beta \neq \beta_m$] $\circ$ [m]
10: $R_{start} :=$ unpackPre(pre)
11: $T :=$ unpackVcond(vcond) $\cup\, T_\varphi$
12: $R := \mathrm{CPA}_b(\mathbb{D}, R_{start}, R_{start})$ // CPA treats b as its CFA
13: $V := \{e \in R \mid [\![e]\!] \cap [\![T]\!] \neq \emptyset\}$
14: $M_{\text{pre}} :=$ packPre$\big(\{e \mid e \in R \wedge e$ located at $l_{exit}\}\big)$
15: $M_{\text{vcond}} :=$ packVcond(V)
16: **broadcast**($M_{\text{pre}} \cup M_{\text{vcond}}$)

and waits for the first message (waitForNextMessage). Once a message $m = (\tau, \beta_m, \cdot)$ arrives, it is updated in the corresponding list: If $\tau = \tau_{pre}$, line 7 removes the previous message of β_m from pre and concatenates m. Analogous, if $\tau = \tau_{vcond}$, line 9 updates vcond.

Line 10 unpacks the preconditions to the set R_{start} of abstract states at the block entry, and line 11 unpacks the violation conditions. The target states T for the current iteration are the unpacked violation conditions and the program specification's original target states T_φ. Then, the algorithm runs $\mathrm{CPA}_b(\mathbb{D}, R_{start}, R_{start})$ with CPA $\mathbb{D}$ on block b. The CPA algorithm starts its analysis with the states in R_{start} by settings them as both the initial reached set and as the initial states in the waitlist. From the computed reached abstract states R, we extract the set V of reached target states (line 13). Then, all reached states at the block exit are packed into M_{pre} (line 14), and all reached target states are packed into M_{vcond} (line 15). The resulting messages are broadcast to all actors (line 16).

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

$$\forall A \subseteq E : \bigcup_{a \in A} [\![a]\!] \subseteq \bigcup_{\alpha \in \text{unpackPre}(\text{packPre}(A, b_0), b_1)} [\![\alpha]\!]$$

The operators packVcond and unpackVcond are sound, iff, for each pair $b_0, b_1 \in B$ of blocks, where b_0 is a predecessor of b_1, and for a set of reached target states V, a block analysis that starts with the communicated violation conditions $R_V = \text{unpackVcond}(\text{packVcond}(V, b_1), b_0)$ finds all of the previously found (concrete) target states $[\![V]\!]$:

$$[\![V]\!] \subseteq [\![CPA_{b_1}(\mathbb{D}, R_V, R_V)]\!]$$

Precision Criteria. The operators packPre and unpackPre are precise, iff 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):

$$\forall A \subseteq E : \bigcup_{a \in A} [\![a]\!] \supseteq \bigcup_{\alpha \in \text{unpackPre}(\text{packPre}(A, b_0), b_1)} [\![\alpha]\!]$$

The operators packVcond and unpackVcond are precise, iff, for each pair $b_0, b_1 \in B$ of blocks, where b_0 is a predecessor of b_1, for a set T of potential target states, and for a set V of previously found target states, a block analysis that starts with $R_V = \text{unpackVcond}(\text{packVcond}(V, b_1), b_0)$ only reaches target states that were reached before (and no additional ones):

$$[\![V]\!] \supseteq [\![CPA_{b_1}(\mathbb{D}, R_V, R_V)]\!] \cap [\![T]\!]$$

Termination Conditions. For each block node in the block graph, we run one separate instance of the DCPA algorithm. The instances are organized in an actor model and communicate messages to all actors. We reach a fix point in one of two cases: (1) When all instances' last broadcast contains no more violation conditions, no more refinements are necessary. In this case, we stop all DCPA instances and claim that program P is safe. (2) When a block with no predecessor broadcasts a message of 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 DCPA instances and claim that program P is unsafe.

Context-Free Instantiation. A context-free instantiation of the DCPA algorithm implements the four operators as follows (with block b, abstract states $E, A \subseteq E, V \subseteq E$):

$$\begin{aligned}
\text{packPre}(A, b) &= \{(\tau_{pre}, b, A_M)\} \\
\text{unpackPre}(M, \cdot) &= \bigcup \{A \mid (\tau_{pre}, \cdot, A_M) \in M\} \\
\text{packVcond}(V, b) &= \{(\tau_{vcond}, b, V_M)\} \\
\text{unpackVcond}(M, \cdot) &= \bigcup \{V \mid (\tau_{vcond}, \cdot, V_M) \in M\}
\end{aligned}$$

The unpack-operators unpack all received abstract states and violation conditions without any modification. This DCPA 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, we find a proof. A violation condition from one function suffices to prove the program unsafe. The behavior is sound but we likely report many false alarms.

Distributed Summary Synthesis. We formulate distributed summary synthesis as an instantiation of the DCPA algorithm. Operator packPre_D (Alg. 3) joins all abstract states into a single abstract state, and unpackPre_D (Alg. 5) unpacks abstract states and stores them in the set states (line 1). Packed states in messages M_{pre} where β is one of the predecessors of b are added to states (lines 2 and 3). If states is empty, we return R_0^b. Otherwise, we return a set containing only the least upper bound of all elements in states.

Operator packVcond_D (Alg. 4) collects all abstract states that violate the specification after traversing the counterexample from the block entry to the violating state $e \in V$ and adds all these states to the known violating states W (line 1). These are sent as violation conditions. Operator unpackVcond_D (Alg. 6) restores the set of known violating states from a set of messages M_{vcond} by collecting all violation conditions V from successors of b. As violations from non-successors are not reached from this block, their violation conditions are skipped.

Algorithm 3 $\text{packPre}_D(E_{in}, b)$

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

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

Algorithm 4 $\text{packVcond}_D(V, b)$

Input: Set V of reached target states, block b
Output: A single message representing all violation conditions for V

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

Algorithm 5 $\text{unpackPre}_D(M_{pre}, b)$

Input: List M_{pre} of messages, block b
Output: Least upper bound {join}

1: states := {}
2: **for** $(\tau_{pre}, \beta, A_M) \in M_{pre}$ **do**
3: **if** $\beta \in$ predecessors(b) **then**
4: states := states $\cup A$
5: **if** states := {} **then**
6: **return** R_0^b
7: join := $\bigsqcup$ states
8: **return** {join}

Algorithm 6 $\text{unpackVcond}_D(M_{vcond}, b)$

Input: List M_{vcond} of messages, block b
Output: Set V_c of violation conditions

1: $V_c := \emptyset$
2: **for** $(\tau_{vcond}, \beta, V_M) \in M_{vcond}$ **do**
3: **if** $\beta \in$ successors(b) **then**
4: $V_c := V_c \cup V$
5: **return** V_c

Soundness of Distributed Summary Synthesis. Operators packPre_D and unpackPre_D are sound:

$$\begin{aligned}
&\text{unpackPre}_D(\text{packPre}_D(E_{in}, b_0), b_1) \\
&= \text{unpackPre}_D(\{(\tau_{pre}, b_0, \{\textstyle\bigsqcup E_{in}\}_M)\}, b_1) \\
&= \textstyle\bigsqcup\{\bigsqcup E_{in}\} \\
&= \textstyle\bigsqcup E_{in}
\end{aligned}$$

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 $\forall e \in E_{in} : e \sqsubseteq \bigsqcup E_{in}$:

$$[\![\textstyle\bigsqcup E_{in}]\!] \supseteq \bigcup_{e \in E_{in}} [\![e]\!].$$

Operators packVcond_D and unpackVcond_D are sound. Let

$$\begin{aligned}
R_V &= \text{unpackVcond}_D(\text{packVcond}(V, b_1), b_0) \\
&= \text{unpackVcond}_D(\{(\tau_{vcond}, b_1, W_M)\}, b_0) \\
&= \bigcup_{e \in V} \omega(\text{cex}(e), e)
\end{aligned}$$

with $b_0, b_1 \in B$ and b_0 is a predecessor of b_1. By construction of the counterexamples, we can find a path from every $e_v \in R_V$ to all $e \in V$. Therefore, $\text{CPA}_{b_1}(\mathbb{D}, R_V, R_V)$ as the CPA explores all reachable states from R_V through the respective counterexample and $V \subseteq \text{CPA}_{b_1}(\mathbb{D}, R_V, R_V)$. This implies the soundness criterion $[\![V]\!] \subseteq [\![\text{CPA}_{b_1}(\mathbb{D}, R_V, R_V)]\!]$.

Precision of Distributed Summary Synthesis. Operators packPre_D and unpackPre_D are not precise, but overapproximate the state-space, as the joins in packPre_D and unpackPre_D produce the least-upper-bound of all abstract states/preconditions.

Operators packVcond$_D$ and unpackVcond$_D$ are precise:

$$[\![V]\!] \supseteq [\![CPA_{b_1}(\mathbb{D}, R_V, R_V)]\!] \cap [\![T]\!].$$

with $R_V = \text{unpackVcond}_D(\text{packVcond}(V, b_1), b_0)$, with $b_0, b_1 \in B$ and b_0 is a predecessor of b_1. It is sufficient to show that every element in $\text{CPA}_{b_1}(\mathbb{D}, R_V, R_V) \cap T$ is contained in V. Let us assume that $e_v \in CPA_{b_1}(\mathbb{D}, R_V, R_V) \cap T$ then, CPA_{b_1} either found a previously known violation V or a potentially new violation $\in T$. For the first case, this holds trivially ($e_v \in V \Rightarrow e_v \in V$). If $e_v \in T$, then we find a subset of restored violation conditions causing e_v: $\omega(\text{cex}(e_v), e_v) \cap R_V \neq \{\}$. The bug must have been uncovered before. Thus, $\omega(\text{cex}(e_v), e_v) \subseteq R_V$ and $e_v \in V$.

Preconditions in Cyclic Block Graphs. With the help of Tarjan's algorithm [9], we identify strongly connected components in the *block graph*. Since originally all preconditions are set to R_0^b, the disjunction of all preconditions in cyclic blocks will repeat the analysis for R_0^b. To avoid this, we apply the following strategy to all blocks in strongly connected components: Whenever a block is part of a strongly connected component, we prevent all preconditions from predecessors that are also part of the same component from being disjuncted unless they are unequal to R_0^b. This ensures that we unroll loops at least once and the precondition of predecessors in the same strongly connected component eventually gets stronger. We can only find valid proofs if all preconditions in the strongly connected component reached a fix point, i.e., $[\![\text{Pre}]\!] \subseteq [\![\text{Pre}']\!]$ where Pre is the precondition of the block and Pre′ is the precondition of the block in the next iteration.

4 EVALUATION

We empirically evaluate our claims using performance experiments. We proposed an approach for software model checking that is based on a decomposition of the verification task into blocks that can be analyzed independently by different threads. Our goal is to reduce the response time, in order to make software model checking feasible for continuous integration.

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 with the response time of our approach using 1, 2, 4, and 8 processing units, to be able to see if the CPU workload is effectively distributed to different 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 our approach using 1, 2, 4, and 8 processing units, to be able to see if the response time is reduced when increasing the number of 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 the new approach with 8 processing units to a standard single-threaded predicate abstraction, to be able to see the potential of the new approach. We select a few verification tasks which employ a sufficient number of workers and block size to see if outperforming award-winning state-of-the-art algorithms is possible.

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 the new approach to the state-of-the-art approaches IMC [18, 50] and k-Induction of CPAchecker [36].

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.

4.2 Experiment Setup

We execute our experiments with version 3.16[2] of BenchExec [51] using an Intel Xeon E3-1230 v5 3.40 GHz processor with 8 processing units and 15 GB 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., [1, 52]).

In our evaluation, we focus on the safe verification tasks of the *SoftwareSystems* category. 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 the individual techniques 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 [53]. The *SoftwareSystems* category contains tasks from real-world programs and is therefore best suitable for our evaluation. This selection defines a benchmark set of 2 485 tasks.

We implement distributed summary synthesis in revision 46372 of the open-source software-verification framework CPAchecker[4] [10]. CPAchecker is implemented in Java with many components readily available. For decomposition of the CFA, we use blk^{merge}. The experimental results are available as reproduction package on Zenodo [53].

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 with 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, 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 [54] states that n cores allow only a maximum speed-up of factor n. Currently, the average, measurable speed-up of *DSS* with eight cores is 4, so half of the theoretical upper bound.

[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

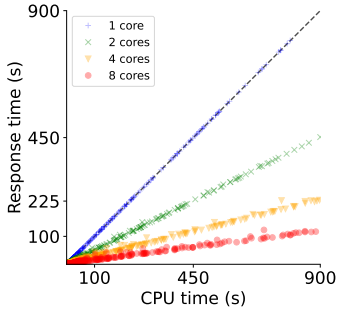

Fig. 6. CPU time (x-axis) and response time (y-axis) for *DSS* with 1, 2, 4, and 8 cores.

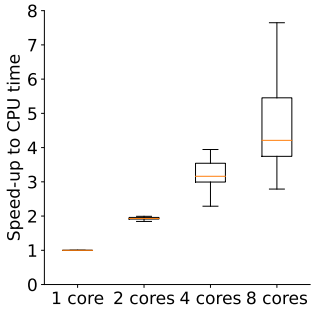

Fig. 7. Speed-up of *DSS* with 1, 2, 4, and 8 cores; the speed-up is the ratio of the CPU time compared to the response time.

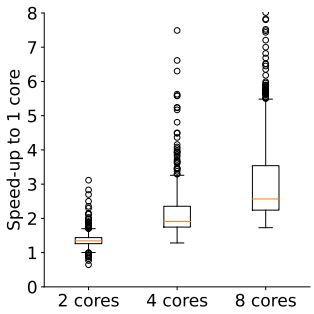

Fig. 8. Speed-up (y-axis) of response time with 2, 4, 8 cores compared to *DSS* with only 1 core; the response time decreases significantly when adding more cores

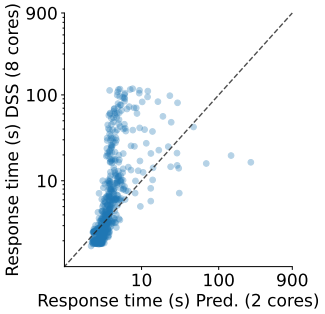

Fig. 9. Comparison of consumed response time of single-threaded predicate abstraction (x-axis) with 2 cores to distributed summary synthesis with 8 cores (y-axis).

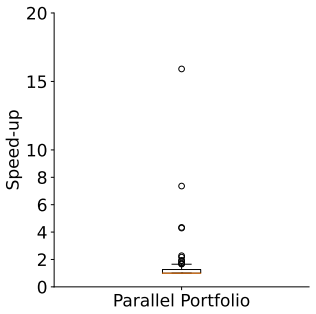

Fig. 10. Speed-up of the parallel portfolio of *DSS* and predicate abstraction compared to standalone predicate abstraction

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 dictated by Amdahl's Law is 2. This is excelled by some tasks because the parallelization changes the behavior of the analysis. SMT queries may

Table 2. Five tasks where distributed summary synthesis (*DSS*) proves tasks significantly faster than predicate abstraction (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	$\text{CPU}_{\mathbb{P}}$	CPU_{DSS}	$\text{RT}_{\mathbb{P}}$	$\mathbf{RT}_{DSS}$	# threads	∅ block size
leds–leds-regulator...	44.76 s	33.23 s	30.83 s	7.18 s	92	12.40
rtc–rtc-ds1553.ko-l...	49.04 s	64.55 s	30.33 s	13.98 s	164	14.98
rtc–rtc-stk17ta8.ko...	46.72 s	67.87 s	28.93 s	15.11 s	162	15.70
watchdog–it8712f_w...	86.78 s	50.26 s	68.99 s	15.89 s	216	7.91
ldv-commit-tester/m0...	50.10 s	102.93 s	28.81 s	20.99 s	230	7.00

change due to the parallel computation of conditions, which can lead to a different, more efficient behavior in the SMT solver.

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 upper 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. 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 2 lists five hand-picked tasks from the benchmark set where distributed summary synthesis outperformed single-threaded predicate analysis. Subscript ℙ shows the data for predicate analysis, subscript *DSS* shows data for distributed summary synthesis.

In these cases, distributed summaries decreases the response time (RT_x) significantly while consuming comparable or more CPU time (CPU_x). 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. 'Unusual' control flow edges as, e.g., `goto`, `continue`, and `break` hinder us from merging blocks, potentially resulting in many small blocks.

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

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 whose weaknesses are not related to the validity of the approach. *DSS* does not complement existing techniques but mainly targets at making 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, they 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 exponentially, 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 parallel portfolio is 16 times faster.

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

Decomposition. 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 define operator packPre_D to run analysis with a join of all preconditions. 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 also define a more precise operator packPre that handles preconditions separately.

Resource Consumption. We spend unnecessary 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). But distributed summary synthesis analyzes blocks in parallel. This means that unnecessary resources are spent on the exploration of a program state space when it is proven unreachable later.

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.

```
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

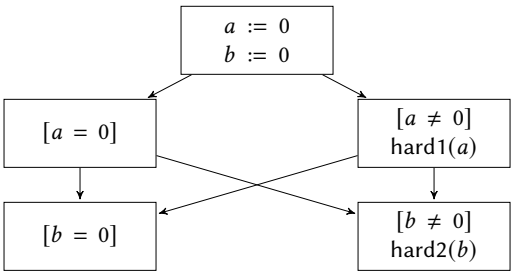

Fig. 12. Poor decomposition of Fig. 11

```
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

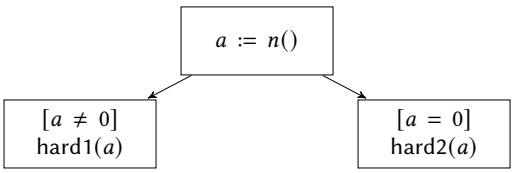

Fig. 14. Good decomposition of Fig. 13

Contrary, if hard blocks are computed in parallel as seen in the decomposed version of Fig. 13 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 error 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 [49] 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 [55]. 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 this work can be underlined with the possibilities it provides 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. Currently, industry often works with generated or handwritten pre- and postconditions of functions. This might become obsolete when using distributed summary synthesis.

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, in principle. 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 [56, 57] to use the best-suiting abstract domain for each block.

Multi-Processing. Since distributed summary synthesis is stateless and communicates only through messages, it allows an easy adaption from multi-threading to multi-processing.

Decompositions. Other program-decomposition strategies may be explored in the future. Similar to AI-based strategy selection for verification techniques [57], we could use AI for finding suiting decompositions of the program.

5 CONCLUSION

This work extends configurable program analysis to a distributed setting (DCPA) and introduces distributed summary synthesis. Distributed summary synthesis is a flexible framework to scale verification through decomposition and distribution. It decomposes a single, large software-verification task into multiple smaller tasks. The parallel and continuously refined synthesis of block summaries 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 [5]. Our experimental results are promising and show that distributed summary synthesis significantly reduces the response time when multiple processing units are available.

Data-Availability Statement. The data from our experiments are available at Zenodo [53] and on our supplementary webpage[5].

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

[5] https://www.sosy-lab.org/research/distributed-summary-synthesis/

REFERENCES

[1] D. Beyer. 2023. Competition on Software Verification and Witness Validation: SV-COMP 2023. In *Proc. TACAS (2) (LNCS 13994)*. Springer.

[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] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] D. Beyer, M. E. Keremoglu, and P. Wendler. 2010. Predicate Abstraction with Adjustable-Block Encoding. In *Proc. FMCAD*. FMCAD, 189–197. https://www.sosy-lab.org/research/pub/2010-FMCAD.Predicate_Abstraction_with_Adjustable-Block_Encoding.pdf.

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] Thomas W. Reps. 1997. Program Analysis via Graph Reachability. In *Proc. ILPS*. MIT, 5–19.

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

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

[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] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. 2011. Certifying Algorithms. *Computer Science Review* 5, 2 (2011), 119–161. https://doi.org/10.1016/j.cosrev.2010.09.009

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

[36] 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

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

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

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

[44] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C.-Henri 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

[45] 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

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

[47] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. 2018. *Handbook of Model Checking*. Springer. https://doi.org/10.1007/978-3-319-10575-8

[48] 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

[49] Carl Hewitt. 2015. Actor Model of Computation: Scalable Robust Information Systems. *arXiv/CoRR* 1008, 1459 (2015). https://doi.org/10.48550/arXiv.1008.1459

[50] D. Beyer, N.-Z. Lee, and P. Wendler. 2022. Interpolation and SAT-Based Model Checking Revisited: Adoption to Software Verification. *arXiv/CoRR* 2208, 05046 (July 2022). https://doi.org/10.48550/arXiv.2208.05046

[51] 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

[52] D. Beyer. 2023. Software Testing: 5th Comparative Evaluation: Test-Comp 2023. In *Proc. FASE (LNCS 13991)*. Springer.

[53] D. Beyer, M. Kettl, and T. Lemberger. 2024. Experimental Results for 'Decomposing Software Verification Using Distributed Summary Synthesis'. Zenodo. https://doi.org/10.5281/zenodo.11095864

[54] 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

[55] 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

[56] 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

[57] 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; accepted 2024-04-16