Domain-Independent Interprocedural Program Analysis using Block-Abstraction Memoization

Anonymous Author(s)

ABSTRACT

Whenever a new software-verification technique is developed, additional effort is necessary to extend the new program analysis to an interprocedural one, such that it supports recursive procedures. We would like to reduce that additional effort. Our contribution is an approach to extend an existing analysis in a modular and domain-independent way to an interprocedural analysis without large changes: We present *interprocedural* block-abstraction memoization (BAM), which is a technique for procedure summarization to analyze (recursive) procedures. For recursive programs, a fix-point algorithm terminates the recursion if every procedure is sufficiently unrolled and summarized to cover the abstract state space.

BAM Interprocedural works for data-flow analysis and for model checking, and is independent from the underlying abstract domain. To witness that our interprocedural analysis is generic and configurable, we defined and evaluated the approach for three completely different abstract domains: predicate abstraction, explicit values, and intervals. The interprocedural BAM-based analysis is implemented in the open-source verification framework CPAchecker. The evaluation shows that the overhead for modularity and domain-independence is not prohibitively large and the analysis is still competitive with other state-of-the-art software verification tools.

ACM Reference Format:
Anonymous Author(s). 2020. Domain-Independent Interprocedural Program Analysis using Block-Abstraction Memoization. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Sacramento, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software verification has been successfully applied to improve the quality and reliability of computer programs [3, 4, 18, 24, 26, 35]. In the last decades, several algorithms and approaches were developed to perform software model checking for various kinds of C programs. However, only a few verifiers for C support full interprocedural analysis, that is, verification of recursive programs: Only 13 out of 22 tool submissions (17 different tools) in the 2020 competition on software verification [6] participated successfully in the benchmark category of recursive tasks.

ESEC/FSE '20, November 8–13, 2020, Sacramento, CA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

A program analysis is called *interprocedural* if procedures are analyzed separately and verification results are merged together from the separate results. The idea is that a program analysis does not depend on long traces through the program, but analyzes procedures independently from each other, such that the result of a procedure's analysis can be used at all call sites with the same context (e.g., with the same abstract arguments). Many verifiers inline called procedures into the calling procedure and verify long traces through a program without any benefit from a modular approach. This not only hinders the reuse of sub-results of the analysis, but also makes it impossible to verify unbounded recursive programs.

We present BAM Interprocedural, a generalization of summary-based interprocedural analysis. The abstract framework is an extension of block-abstraction memoization (BAM) [10, 49].

Example. We outline how to prove the correctness of the example program in Fig. 1 (illustrated in Fig. 2), which uses two unsigned integer variables a and b, and nondeterministically initializes them as input for the recursive procedure *sum* that returns the sum of its arguments. The program is deemed correct if *error*() is not called.

This program can not be verified by a default bounded model checker that iteratively unrolls the recursion, because the number of unrollings is unknown. However, using a procedure summary like $ret = m + n$, where m and n are the parameters of procedure *sum* and ret is the return value of the procedure call, would help with the verification. This summary is a valid abstraction for the control-flow for every call of the procedure *sum* and can be applied as a substitution for the initial call in procedure *main* as well as for the recursive call in procedure *sum* itself. For a fully automated analysis, the verification algorithm must come up with this (or some similar) summary and apply it as part of the proof strategy.

This example program requires an abstract domain that tracks relations between variables. Thus, a standard predicate analysis (such as in Sect. 4) is able to infer such predicates (e. g., via CEGAR [23] and interpolation [37]) and can soundly apply procedure summaries for all call-sites of a procedure. In general, our approach works on a domain-independent level and does not depend on SMT-based summaries. The combination of procedure summaries with a fixed-point algorithm computes an over-approximation of the reachable state space of the recursive procedure. The algorithm first determines a procedure summary for a single unrolling of the procedure, i.e., for all paths through the procedure that are not traversing the recursive call. Using the above mentioned abstract domain, the analysis obtains a summary like $ret = m + n$ in this first step. Then, the algorithm applies the computed procedure summary to the recursive call and explores longer paths through the program and refines the procedure summary until the algorithm cannot explore any new path. For the given example, applying this summary once for the recursive procedure call within the procedure *sum* does not change the summary of the whole procedure *sum*, thus it is sufficient to reach a fixed-point and the analysis can terminate.

```
1  void main(void) {
2    uint a = nondet();
3    uint b = nondet();
4    uint s = sum(a, b);
5    if (s != a + b) {
6      error();
7    }
8  }
9
10 uint sum(uint n, uint m) {
11     if (n == 0) {
12       return m;
13     } else {
14       uint tmp = sum(n - 1, m + 1);
15       return tmp;
16     }
17 }
```

Figure 1: Example program with a recursive procedure *sum*

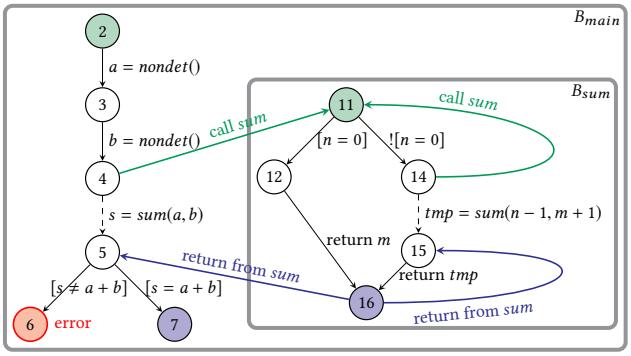


Figure 2: CFA for the example program in Fig. 1, with procedures B_{main} and B_{sum}

Contribution. Our contribution consists of three parts:

(1) We present a *domain-independent* approach of BAM [49] for a fully interprocedural analysis: every procedure is analyzed separately and the result of a procedure's analysis (an abstraction of the procedure, also known as "procedure summary") is integrated in the analysis of the calling context.

(2) A program might contain *unbounded recursion* (e.g., the recursion depth is depending on unknown input). Instead of just cutting off program traces at a predefined depth, our analysis terminates the unrolling of a recursive procedure in a sound way once a fixed point is reached, and does not omit feasible error paths. The fixed-point algorithm iteratively increments the unrolling of the recursion until no new abstract state is reachable. The algorithm is domain-independent, because only coverage checks for abstract states are used, which are already provided by each abstract domain. The overhead is negligible for non-recursive programs.

(3) We formally define an *additional domain-specific operator* rebuild in the framework, such that recursive procedures can be handled in every domain. This operator restores eliminated information of the calling context after leaving a recursive call.

Availability of Data and Implementation. All benchmark tasks for evaluation, configuration files, a ready-to-run version of our implementation, and tables with detailed results are available in the artifact [2]. The source code of our extensions to the open-source verification framework CPAchecker [13] is available in the project repository; see https://cpachecker.sosy-lab.org.

Related Work. As programs with (recursive) procedures have been analyzed and also verified since decades, many ideas are already available and implemented in some tools. We give a short overview of the tools and the domains they are based on.

Inlining-Based Analysis. A common approach to analyze procedures in bounded model checking is to unroll them up to a certain limit and ignore any deeper recursive calls. Tools like Cbmc [25], Esbmc [31], and Smack [41] implement this approach, which leads to an unsound analysis in combination with recursive procedure calls, because there is no guarantee that the bug is unreachable through further unrolling. Without the user specifying a bound, the model checker might run into an endless unrolling of the recursion. Constant propagation (like in Cbmc) or additional checks can avoid too far unrolling of recursive procedures. Also unbounded frameworks like CPAchecker [13] have several analyses based on different domains [14, 15, 40] that inline procedure calls. Our approach is build on top of them and reuses existing components, such that the amount of changes to a single analysis is minimal.

Interpolation-Based Summaries. There are also interpolation-based [28] approaches to verify recursive programs. The bounded model checker FunFrog [46, 47] generates interpolation-based procedure summaries to avoid the repeated analysis of procedures. Whale [1] is an extension of Impact [38] and analyzes recursive procedures using two types of formulas in its intra-procedural analysis, namely state- and transition-interpolants, to get summaries. Those approaches separately analyze each procedure until a fixed-point is reached and the procedures (or the representing formulas) are sufficiently refined. Ultimate Automizer uses nested interpolants [33] to compute formulas for procedures depending on the caller's context. Those approaches are bound to an SMT-based domain and the algorithms do not support combinations with other domains.

Further Domain-Specific Interprocedural Analyses. Bebop [5] computes procedure summaries for boolean programs. The application of Bebop however is limited to boolean programs and abstract states are described with binary decision diagrams. Abductor [19] is an interprocedural program verifier that applies the domain of separation logic to prove memory-related safety properties. Additionally, a recursive program can be transformed into a non-recursive one, such that any verification tool without direct support for recursion can be used indirectly to analyze the recursive program. For example, CPArec [22] is a light-weight approach using an external black-box verifier and a fixed-point algorithm that increments the unrolling depth to compute procedure summaries until coverage is reached. This approach is limited to predicate-based verifiers.

Interprocedural Data-Flow Analysis. The above examples are based on symbolic analysis, i.e., depending on BDD-, SAT-, or SMT-based domains, while our proposed approach works for classic data-flow domains as well. The classic approach to interprocedural data-flow analysis [42] is restricted to finite-height lattices of domain elements and an operator yielding the join of two domain elements.

BAM Interprocedural works for arbitrary, unlimited abstract domains and different operators for combining elements (depending on the represented data, not only join) or coverage checks for elements (domain-specific comparison).

2 BACKGROUND

We describe the program representation as a control-flow automaton and domain-independent reachability analysis based on the concept of configurable program analysis. Afterwards, their application as components in an interprocedural analysis is shown.

2.1 Programs

A *program* is represented by a *control-flow automata* (CFA) $A = (L, l_0, G)$ that consists of a set L of program locations, an initial program location $l_0 \in L$, and a set $G \subseteq L \times Ops \times L$ of control-flow edges. An edge models the control-flow operation (from *Ops*) between program locations, for example assignments or assumptions. Figure 2 represents the example program as CFA. Our presentation uses a simple imperative programming language, which allows only assignments, assume operations, procedure calls and returns, and all variables are integers. The implementation of our tool provides basic support for heap-related data-structures including pointers and arrays, but this description avoids them for simplicity.

2.2 Blocks in a Program

Blocks are formally defined as parts of a program: A block $B = (L', G')$ of a CFA $A = (L, l_0, G)$ consists of a set $L' \subseteq L$ of program locations and a set $G' = \{(l_1, op, l_2) \in G \mid l_1, l_2 \in L'\}$ of control-flow edges. We assume that two blocks B and B' are either disjoint ($B.L' \cap B'.L' = \emptyset$) or one block is completely nested in the other block ($B.L' \subset B'.L'$). Each block has *input* and *output locations*, which are defined as $In(B) = \{l \in L' \mid (\exists l' : (l', \cdot, l) \in G \wedge l' \notin L') \vee (\nexists l' : (l', \cdot, l) \in G)\}$ and $Out(B) = \{l \in L' \mid (\exists l' : (l, \cdot, l') \in G \wedge l' \notin L') \vee (\nexists l' : (l, \cdot, l') \in G)\}$, respectively. In general, the block size can be freely chosen in our approach. For an interprocedural analysis, we use procedures as blocks, such that a block abstraction represents a procedure summary. In Fig. 2, the blocks B_{main} and B_{sum} represent the two procedures of the program. The input and output locations are marked for each block.

2.3 CPA and CPA algorithm

The reachability analysis is based on the concept of configurable program analysis (CPA) [12], which specifies the abstract domain for a program analysis and additional operations.

A CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$ consists of an abstract domain D, a transfer relation $\rightsquigarrow$, and the operators merge and stop. The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set C of concrete states, a semi-lattice $\mathcal{E} = (E, \sqsubseteq)$ over a set E of abstract-domain elements (i.e., abstract states) and a partial order $\sqsubseteq$ (the join $\sqcup$ of two elements and the join $\top$ of all elements are unique), and a concretization function $[\![\cdot]\!] : E \to 2^C$ that maps each abstract-domain element to the represented set of concrete states. The transfer relation $\rightsquigarrow \subseteq E \times E$ computes abstract successor states, a transfer relation $\overset{g}{\rightsquigarrow}$ matches the transfer along an edge $g \in G$ of the CFA. The merge operator $\text{merge} : E \times E \to E$ specifies if and how to merge two abstract states when control flow meets. The stop operator $\text{stop} : E \times 2^E \to \mathbb{B}$ determines whether an abstract state is covered by a given set of abstract states. The operators merge and stop can be chosen appropriately to influence the abstraction level of the analysis. Common choices include $\text{merge}^{sep}(e, e') = e'$ (which does not merge abstract states) and $\text{stop}^{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$ (which determines coverage by checking whether the given abstract state is less than or equal to any other reachable abstract state according to the semi-lattice).

Given a CPA, we can apply a reachability algorithm (denoted as CPA algorithm in [12]) that explores the abstract state space of a program and computes all reachable abstract states. The stop operator determines the fixed-point criteria, i. e., whether a state has already been discovered before. For the following description, we consider a reachability analysis $CPA(\mathbb{D}, \text{reached}, \text{waitlist})$ using a CPA $\mathbb{D}$ and two sets reached and waitlist of abstract states as input and returning two sets reached′ and waitlist′ of abstract states. The idea is that starting with the given sets of already reached abstract states and a frontier waitlist, the reachability algorithm computes more reachable successors and a new frontier waitlist.

The CPA algorithm can be used as component in a CEGAR-based fixed-point loop [23] to refine the granularity of the current analysis. For simplicity we ignore the precision in the description.

In the following Sect. 3, we describe our interprocedural extension of block-abstraction memoization, and then in Sect. 4 provide an application of the concept to three separate domains: the Callstack-CPA for tracking a call stack of the program, the Predicate-CPA for handling variable assignments with predicates, and the Value-CPA for tracking variable assignments explicitly.

3 BAM FOR INTERPROCEDURAL ANALYSIS

Block-Abstraction Memoization (BAM) [49] is a modular and scalable approach for model checking abstract state spaces by leveraging the idea of *divide and conquer*. BAM divides a large program into smaller parts, named *blocks*, and analyzes them separately. The result of a block's analysis is denoted as a *block abstraction*. Block abstractions are stored in a cache. Whenever a larger block depends on a nested block, a block abstraction of the nested block is created during the larger block's analysis. Block abstractions are independent of a concrete domain and work on an abstract level. There can be several block abstractions for the same block, e. g., depending on different input values of the block.

In the following, we use procedures as blocks. More precisely, a *procedure block* B_f consists of the procedure f itself and all procedures that are (transitively) called from f, such that the whole control-flow of nested blocks, including call and return edges are included in the block B_f (see Fig. 2).

BAM ensures efficiency by using a cache $\text{cache} \subseteq (Blocks \times E) \to (2^E \times 2^E)$ for block abstractions, which maps the initial abstract state for a block to the block abstraction. The block abstraction is defined as the set of reached abstract states and the set of frontier abstract states, which both are computed during the block's analysis.

BAM is defined recursively (independent of any recursion in the analyzed program) and repeatedly (nestedly) applies the reachability analysis. Our implementation of BAM uses a *stack* of pairs $p \in Blocks \times E$ that consists of all currently open analyses referenced by their block of the CFA to be analyzed and an initial abstract state (starting point of the block abstraction).

This section defines BAM Interprocedural. We show that procedure blocks correspond to procedure summaries, describe the problems of analyzing recursive procedures, the necessity of the fixed-point algorithm, and a new operator rebuild.

3.1 Operators of BAM

BAM uses two complementing operators reduce $\subseteq Blocks \times E \to E$ and expand $\subseteq Blocks \times E \times E \to E$, and an additional operator rebuild $\subseteq E \times E \times E \to E$, to drop or restore context-based information for each analyzed block. On an abstract level, the reduce operator performs an abstraction of the given abstract state and the expand operator concretizes an abstract state for a given context. These operators aim towards an interprocedural analysis where each block can be analyzed without knowing its concrete context. How much of this context-independence can be achieved depends on the concrete domain (see Sect. 4 for more details). The implicit benefit of the first two operators is an improvement of the cache-hit-rate. The operator reduce drops unimportant information from an abstract state when entering a block. The resulting abstract state is more abstract and is used as cache key and as initial abstract state for the block's analysis. The importance of some information depends on the wrapped analysis and the available block. For example, variables, predicates, or levels of the call stack that are not accessed inside the entered block, but only depend on the surrounding context, might be good candidates to be removed from the abstract state. The operator expand restores removed information for abstract states when applying the block abstraction in the surrounding context. The operator rebuild avoids collisions of program identifiers (like variables) when returning from a (possibly recursive) procedure scope into its calling context. This operator does not compute an abstraction, but performs simple operations depending on the given abstract domain such as renaming variables, substituting predicates, or updating indices.

With these operators, we now formally define the CPA for BAM.

3.2 BAM as CPA

For usage with the CPA concept (see Sect. 2.3), BAM itself is formalized as a CPA $\mathbb{BAM} = (D_{\mathbb{BAM}}, \rightsquigarrow_{\mathbb{BAM}}, \text{merge}_{\mathbb{BAM}}, \text{stop}_{\mathbb{BAM}})$. As BAM works on an abstract, domain-independent level, it requires a separate abstract-domain-dependent analysis (like the *value analysis* or *predicate analysis*) to track variables, values, and assignments. This separate analysis is also defined via the CPA concept (see Sect. 4) and for the following formalization we denote it as a general (wrapped) CPA $\mathbb{W} = (D_{\mathbb{W}}, \rightsquigarrow_{\mathbb{W}}, \text{merge}_{\mathbb{W}}, \text{stop}_{\mathbb{W}})$.

(1) The domain $D_{\mathbb{BAM}}$ is the wrapped domain $D_{\mathbb{W}}$, i. e., BAM simply uses the abstract states of the underlying domain.

(2) The transfer relation includes the transfer $e \rightsquigarrow_{\mathbb{BAM}} e'$ for two abstract states e and e' and a block B if

$$e' \in \begin{cases} \mathit{fixedPoint}(B_{main}, l, e) & \text{if } l = l_0 \text{ and } \mathit{stack} = [] \\ \mathit{applyBlockAbstraction}(B, e) & \text{if } l \in \mathit{In}(B) \\ \{e'' \mid e \rightsquigarrow_{\mathbb{W}} e''\} & \text{if } l \notin \mathit{Out}(B) \end{cases}$$

where l is the program location for e and *stack* is the internal stack of nested blocks during the analysis.
The transfer relation applies one of three possible steps: (1) The fixed-point algorithm Alg. 1 is executed if the current program location is the initial program location l_0 and the *stack* is empty. (2) At an input location of a block B, i. e., if a new nested block would be entered from a surrounding context, we apply the block abstraction returned from the operation *applyBlockAbstraction* (cf. Alg. 2) for the nested block. (3) For output locations of blocks, there is no succeeding abstract state (in the sub-analysis). For other program locations, the wrapped transfer relation $\rightsquigarrow_{\mathbb{W}}$ is applied.

(3) The merge operator $\text{merge}_{\mathbb{BAM}} = \text{merge}_{\mathbb{W}}$ delegates to the wrapped analysis, i. e., BAM merges whenever the underlying domain merges abstract states.

(4) The termination check $\text{stop}_{\mathbb{BAM}} = \text{stop}_{\mathbb{W}}$ delegates to the wrapped analysis, i. e., the coverage relation between abstract states depends on the underlying domain.

Algorithm 1 *fixedPoint*(B_{main}, l_0, e_0)

Input: block B_{main} with initial program location l_0, abstract state e_0
Output: set of reachable states, which all represent output states of the block B_{main}
Global Variables: boolean flag fixedpointReached
Variables: set blockResult of abstract states

```
1: repeat
2:     fixedpointReached := true;
3:     blockResult := applyBlockAbstraction(B_main, e_0);
4: until fixedpointReached
5: return blockResult;
```

Algorithm 2 *applyBlockAbstraction*(B, e_I)

Input: abstract state e_I at a block input location of a block B
Output: abstract states for the output locations of the analyzed block B
Global Variables: boolean flag fixedpointReached,
set cache mapping a block and an abstract state to a block abstraction,
sequence stack consisting of pairs of a procedure block and an abstract state
Variables: sets reached and waitlist of abstract states for the analysis of the current block

```
1: e_i := reduce_W(B, e_I);
2: if ∃(B, e_c) ∈ stack : e_i ⊑ e_c then
3:     if cache contains (B, e_c) then
4:         reached, _ := cache(B, e_c);
5:     else
6:         reached := {}
7:         fixedpointReached := false;
8: else
9:     if cache contains (B, e_i) then
10:        reached, waitlist := cache(B, e_i)
11:    else
12:        reached := {e_i}; waitlist_r := {e_i}
13:    stack.push((B, e_i));
14:    reached, waitlist := CPA(W, reached, waitlist)
15:    stack.pop();
16:    if cache contains (B, e_i) then
17:        reached_old, _ := cache(B, e_i);
18:        for e ∈ reached do
19:            if loc(e) ∈ Out(B) ∧ ∄e' ∈ reached_old : e ⊑ e' then
20:                fixedpointReached := false;
21:    cache(B, e_i) := (reached, waitlist)
22: e_call := getPredecessor(e_I);
23: tmp := {expand_W(B, e_I, e_o) | e_o ∈ reached ∧ loc(e_o) ∈ Out(B)}
24: return {rebuild_W(e_call, e_I, e_O) | e_O ∈ tmp};
```

The transfer relation $\rightsquigarrow_{\mathbb{BAM}}$ uses the fixed-point algorithm and the computation of block abstractions as explained in the next subsections.

3.3 Fixed-Point Algorithm for Unbounded Recursion

An analysis of recursive procedures must handle a possibly unbounded unrolling of the call stack if the information of an abstract state is insufficient to avoid deeper exploration and can not cut off the state space. In our approach, the fixed-point algorithm (*fixedPoint*, Alg. 1) repeatedly analyzes the program using *applyBlockAbstraction* (Alg. 2) from the initial program location onwards. It iteratively increments the number of unrollings and terminates only if coverage was reached for all analyzed procedure calls.

In each iteration of the fixed-point algorithm, we generate an overapproximation of some (more) paths through the recursive procedure (because of the limited unrolling of the recursion) and determine a summary for the currently analyzed procedure block. The termination is decided by a coverage check for the abstract states of the analyzed block summary.

The first iteration of the fixed-point algorithm assumes no valid path through the recursive call. We only explore the non-recursive parts of the program's control flow and skip the recursive call of the procedure. Depending on the abstract domain, the initial summary for the recursive procedure is an empty set of abstract states (Alg. 2, line 6). The block abstraction of a procedure is stored in the cache after returning from the procedure call (Alg. 2, line 21). In further iterations, we increment the limit of unrollings of the recursive procedure and refine the block abstraction, analyze the program again, starting from the initial program location (and using several intermediate results from the cache), until the procedure summary becomes stable.

3.4 Soundness of BAM for Recursion

The fixed-point criteria are based on Hoare's rule for recursion (Fig. 3): if the body of a procedure f satisfies the pre- and post-conditions P and Q (including parameter passing and return values) under the condition that all recursive calls to the procedure f satisfy P and Q, then the whole procedure f satisfies P and Q. Translated into our model, we use (concretizations of) abstract states as pre- and post-conditions of statements, the procedure and its body corresponds to the procedure's block; Fig. 4 shows the resulting rule. The renaming (or an equivalent operation) of equal identifiers from the recursive call of f, which appear in the calling and called procedure f, is shifted into a different part of the analysis (see Sect. 3.5 on operator rebuild) and is handled in a sound way.

To determine the fixed-point criteria for termination, Alg. 2 checks the following two properties during the analysis.

Firstly, we try to stop the unrolling of an unbounded recursive procedure by an over-approximating analysis. Thus, before analyzing a new recursive procedure call, we check whether the abstract state at a procedure entry is already covered by any abstract state from the current stack (Alg. 2, line 2). If such a covering abstract state exists, we skip the recursive call and use a procedure summary instead of further exploring the recursive call (Alg. 2, line 3 to 7).

$$\frac{\{P\}b = f(a)\{Q\} \quad \vdash \quad \{P \wedge p = a\}B_f\{Q \wedge p = a \wedge b = r\}}{\{P\}b = f(a)\{Q\}}$$

Figure 3: Hoare's rule for recursion, for a given procedure definition $f(p)\ \{B_f;\ return\ r;\}$

$$\frac{\{[\![P_e]\!]\}b = f(a)\{[\![Q_e]\!]\} \quad \vdash \quad \{[\![P_e]\!]\}B_f\{[\![Q_e]\!]\}}{\{[\![P_e]\!]\}b = f(a)\{[\![Q_e]\!]\}}$$

Figure 4: Hoare's rule for recursion (with abstract states)

The procedure summary consists of either previously computed abstract successor states from the BAM cache or (in case of a cache miss) no successor states at all.

Secondly, because a procedure summary represents only a bounded execution of the called procedure, this approach alone represents only a subset of possible traces in the procedure and might be unsound in cases that require deeper unrolling. To determine if the inserted procedure summaries are sufficient for Hoare's rule of Fig. 4, we check for coverage of the exit state (of the procedure executed with the inserted procedure summary) against the previously computed abstract states (of the procedure summary). This check is performed in lines 18 to 20 of Alg. 2. If the coverage relation is satisfied (for all procedures in the program), then the fixed-point algorithm terminates, because fixedpointReached is never set to *false* during the iteration. In this case we have found a sound over-approximation of the recursive procedure. Otherwise the fixed-point algorithm continues.

3.5 Block-Abstraction Computation with Operators

The operation *applyBlockAbstraction* (cf. Alg. 2) starts with the reduction $\text{reduce}_{\mathbb{W}}(B, e_I)$ of initial abstract state e_I and determines the block abstraction for a block B. The block abstraction is either taken from the cache or computed via a separate application of the reachability algorithm (i. e., CPA algorithm). To integrate the block abstraction into a surrounding context, the operators $\text{expand}_{\mathbb{W}}$ and $\text{rebuild}_{\mathbb{W}}$ are applied to each abstract state at the block's output location. The operators reduce and expand abstract or concretize the given abstract state and aim to increase the cache-hit rate of BAM. For an interprocedural approach, they remove and restore (most of) the context-based information for a procedure block.

While the fixed-point algorithm handles over-approximations and refinements of block abstractions, an interesting detail of the implementation remains open: How can we identify and work with symbols, i.e., variable identifiers, across procedure scopes? Identical identifiers for program variables of the same procedure scope are problematic for the analysis of recursive procedures. Due to the modularity of the framework CPAchecker, only a separate call-stack analysis knows about procedure scopes and all other analyses assume unique identifiers across all operations. BAM also tracks information about procedures in its stack, but it does not use this information for detailed analysis of variables and identifiers. Each recursive procedure entry starts a new procedure scope, where the

identifiers override existing (valid) identifiers from previous call-stack levels. Entering a procedure and overriding existing identifiers from the calling scope is no problem, because only the most local version of an identifier is available (and visible) in the procedure scope. Leaving the procedure afterwards is more complex, because identifiers are overridden during the procedure's traversal and have to be restored to match the calling context.

A solution like a simple renaming of identifiers is not possible, because each domain has its own way of representing variables. Additionally, each domain must have a strategy for handling scoped variables that allows a consistent use of the cache in BAM.

We solve this problem by using a new operator rebuild : $E \times E \times E \rightarrow E$, and we show how to implement it for different domains. The operator rebuild is applied after analyzing the procedure-exit location (Alg. 2, line 24), i.e., after leaving the block of a (maybe recursive) procedure and after the application of the operator expand. The operator rebuild maps three abstract states (information about the calling context from the procedure call state e_{call}, information about the arguments and parameters of the called procedure from the procedure entry state e_I, and information about the return value and the block abstraction from the procedure exit state e_O) to a new abstract state that is a successor of the procedure call and a valid starting point for the further analysis. The operator rebuild is defined depending on the underlying analysis.

4 APPLICATION OF INTERPROCEDURAL BAM TO CONCRETE DOMAINS

In this section, we describe some concrete analyses that can be used in the context of BAM and provide a way of computing a (nearly) context-independent block abstraction. Using the framework CPAchecker, program analyses can be composed of several component CPAs. Separate CPAs are defined and implemented for tracking the program counter, the predecessor-successor relationship of the reachability graph, or for combining other CPAs in a composite analysis. Thus, we do not need to specify these aspects when defining a core analysis, but directly specify the concrete analyses. In the following, we explain an analysis for tracking the call stack and two more precise analyses for analyzing variables and assignments (namely *value analysis* and *predicate analysis*). The analyses can be used as component in a CEGAR-loop and thus their refinement strategies will be considered.

Callstack-CPA. The CPA for *call-stack analysis* $\mathbb{C} = (D_\mathbb{C}, \rightsquigarrow_\mathbb{C}, \text{merge}_\mathbb{C}, \text{stop}_\mathbb{C})$ explicitly tracks the call stack $s = [f_1, \cdots f_n]$ of the program, where f_1 to f_n denote procedures scopes for an abstract state s.

(1) The domain $D_\mathbb{C} = (C, \mathcal{E}_\mathbb{C}, [\![\cdot]\!])$ is based on the flat lattice $\mathcal{E}_\mathbb{C} = (S \cup \{\top\}, \sqsubseteq)$ for the set S of possible call stacks. The expression $s \sqsubseteq s'$ is fulfilled if $s = s'$ or $s' = \top$, $[\![\top]\!] = C$. For all s in S, we have $[\![s]\!] = \{c \in C \mid \text{callstackOf}(c) = s\}$.

(2) The transfer relation $\rightsquigarrow_\mathbb{C}$ has the transfer $s \overset{g}{\rightsquigarrow}_\mathbb{C} s'$ for an abstract state $s = [f_1, \cdots, f_{n-1}, f_n]$ and

$$s' = \begin{cases} [f_1, \cdots, f_n, f_{n+1}] & \text{if } g \text{ is a procedure call to } f_{n+1} \\ [f_1, \cdots, f_{n-1}] & \text{if } g \text{ is a procedure return from } f_n \\ s & \text{otherwise} \end{cases}$$

(3) The merge operator $\text{merge}_\mathbb{C} = \text{merge}^{sep}$ does not combine abstract states.

(4) The termination check $\text{stop}_\mathbb{C} = \text{stop}^{sep}$ returns whether the same abstract state was already reached before.

(5) The reduce operator $\text{reduce}_\mathbb{C}$ abstracts from a concrete call stack and keeps only the context-relevant suffix. Therefore, it determines the maximum procedures scope range of the current block, i. e., how many procedures symbols we can pop from the current call stack $[f_1, ..., f_i, ..., f_n]$ during an analysis of the current block. Let the procedures symbol f_i be the minimal procedure scope reachable during the block's analysis. Then, the operator keeps only the reachable (most local) procedure scopes from the abstract state: $\text{reduce}_\mathbb{C}(B, [f_1, ..., f_i, ..., f_n]) = [f_i, ..., f_n]$.

(6) The expand operator $\text{expand}_\mathbb{C}$ restores the removed part of the call stack: $\text{expand}_\mathbb{C}([f_1, ..., f_i, ..., f_n], B, [f_i, f_j, ..., f_s]) = [f_1, ..., f_i, f_j, ..., f_s]$.

(7) The *call-stack analysis* does not track variables, but only the procedure scope itself. Thus the rebuild operator is defined as $\text{rebuild}_\mathbb{C}(e_{call}, e_I, e_O) = e_O$.

Value-CPA. The CPA for *value analysis* $\mathbb{E} = (D_\mathbb{E}, \rightsquigarrow_\mathbb{E}, \text{merge}_\mathbb{E}, \text{stop}_\mathbb{E})$ explicitly tracks the assignments of variables. The CPA is used as described in previous work [15] and extended with the additional operators reduce and expand.

(1) The domain $D_\mathbb{E} = (C, \mathcal{E}_\mathbb{E}, [\![\cdot]\!])$ is based on the lattice $\mathcal{E}_\mathbb{C} = (V, \top_\mathbb{E}, \sqsubseteq_\mathbb{E}, \sqcup_\mathbb{E})$ for the set $V = (X \rightarrow \mathcal{Z})$ of abstract variable assignments with a set X of variables and a set $\mathcal{Z} = \mathbb{Z} \cup \{\top_\mathcal{Z}, \bot_\mathcal{Z}\}$ of values ($\mathbb{Z}$ denotes the set of integer values). We use the (flat) lattice over numbers with the least upper bound $\top_\mathcal{Z}$ and the greatest lower bound $\bot_\mathcal{Z}$. Let $v(x)$ be the value of a variable $x \in X$ in the assignments of $v \in V$. The top element $\top_\mathbb{E} \in V$ denotes the abstract variable assignment with no specific value for any variable ($\forall x \in X : \top_\mathbb{E}(x) = \top_\mathcal{Z}$). The partial order $v \subseteq V \times V$ is defined as $v \sqsubseteq v'$ if $v(x) = v'(x)$, $v(x) = \bot_\mathcal{Z}$, or $v'(x) = \top_\mathcal{Z}$ is satisfied for all $x \in X$. The concretization function $[\![\cdot]\!] : V \rightarrow 2^C$ returns the meaning for each abstract data state v.

(2) The transfer relation $\rightsquigarrow_\mathbb{E}$ has the transfer $v \overset{g}{\rightsquigarrow}_\mathbb{E} v'$ for an abstract state v, if

(1) $g = (\cdot, assume(exp), \cdot)$ and for all $x \in X$:

$$v'(x) = \begin{cases} \bot_\mathcal{Z} & \text{if } (y, \bot_\mathcal{Z}) \in v \text{ for some } y \in X \text{ or} \\ & \text{the assumption } exp_{/x} \text{ is unsatisfiable} \\ c & \text{if } c \text{ is the only satisfying assignment of ,} \\ & \text{the assumption } exp_{/x} \text{ for the variable } x \\ \top_\mathcal{Z} & \text{otherwise} \end{cases}$$

or

(2) $g = (\cdot, w := exp, \cdot)$ and for all $x \in X$:

$$v'(x) = \begin{cases} exp_{/v} & \text{if } x = w \\ v(x) & \text{if } x \in def(v), \\ \top_\mathcal{Z} & \text{otherwise} \end{cases}$$

where the evaluation $exp_{/v}$ of an expression exp with an abstract state $v \in V$ is defined as the substitution of all variables x in exp with their values $v(x)$ and interpreting the result.

(3) The merge operator $\text{merge}_{\mathbb{E}} = \text{merge}^{sep}$ does not combine abstract states.
(4) The termination check $\text{stop}_{\mathbb{E}} = \text{stop}^{sep}$ returns whether a covering abstract state was already reached before.
(5) The reduce operator $\text{reduce}_{\mathbb{E}}$ only keeps assignments of variables that are accessed in the block's context: $\text{reduce}_{\mathbb{E}}(B, e_I) = \{(x, e_I(x)) \in e_I \mid x \text{ used in } B\}$.
(6) The expand operator $\text{expand}_{\mathbb{E}}$ restores the assignments that were removed with $\text{reduce}_{\mathbb{E}}$ from the initial abstract state: $\text{expand}_{\mathbb{E}}(e_I, B, e_o) = \{(x, e_I(x)) \in e_I \mid x \text{ not used in } B\} \cup e_o$.
(7) We define *global* variables as symbols declared in global scope and the rest as *local* variables, i.e. symbols declared in local procedure scope. After leaving a (recursive) procedure call, the operator $\text{rebuild}_{\mathbb{E}}$ considers local variables from the calling scope, and global variables and the return value from the exited procedure scope: $\text{rebuild}_{\mathbb{E}}(e_{call}, e_I, e_O) = \{(x, v) \in e_{call} \mid \neg isGlobal(x) \wedge \neg isReturn(x)\} \cup \{(x, v) \in e_O \mid isGlobal(x) \vee isReturn(x)\}$.
Because global variables can be assigned during the procedure's execution, they are not reset to their assigned value from before the procedure's execution; their values are also taken from the abstract state e_O at the procedure's exit location.

Note that, with these definitions of $\text{reduce}_{\mathbb{E}}$ and $\text{expand}_{\mathbb{E}}$, the *value analysis* is not completely context-independent and interprocedural, because a block abstraction for this domain depends on the input values of variables accessed in the block. For procedure blocks, we are independent of the surrounding context and a block abstraction for a function call can be applied whenever function arguments have identical values.

Predicate-CPA. The CPA for *predicate analysis* $\mathbb{P} = (D_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \text{merge}_{\mathbb{P}}, \text{stop}_{\mathbb{P}})$ uses predicates to track variables and their values [9, 49]. For this analysis a set $\mathcal{P}$ of predicates is used, which can be incrementally computed in CEGAR loop [23] that is applied on top of the CPA algorithm. In this description, we do not go into detail on how to determine concrete predicates, but just assume that the predicates are already available, e. g., by applying a refinement strategy as defined in [11, 14].

The refinement procedure of the *predicate analysis* computes interpolants that match the structure of the procedure blocks [33] and allow an interprocedural analysis.

For each block B, we divide the set $\mathcal{P}$ of predicates into two disjoint partitions $\mathcal{P}_B = \{p \in \mathcal{P} \mid p \text{ used in } B\}$ and $\mathcal{P}_{\neg B} = \mathcal{P} \setminus \mathcal{P}_B$. The partition $\mathcal{P}_B$ contains all predicates from $\mathcal{P}$ that reference variables of the block, e. g., these predicates reference variables accessed in the block. The partition $\mathcal{P}_{\neg B}$ contains the rest of $\mathcal{P}$.

(1) The domain $D_{\mathbb{P}} = (C, \mathcal{E}_{\mathbb{P}}, [\![\cdot]\!])$ is based on the lattice $\mathcal{E}_{\mathbb{P}} = (E, \top_{\mathbb{P}}, \sqsubseteq_{\mathbb{P}}, \sqcup_{\mathbb{P}})$ and a concretization function $[\![\cdot]\!] : E \to C$. The lattice consists of abstract states $e \in E$, which are tuples $(\psi, l^{\psi}, \varphi) \in (\mathcal{P} \times (L \cup \{l_\top\}) \times \mathcal{P})$. The *state formula* ψ is a boolean combination of predicates from $\mathcal{P}$ and has been computed at the program location l^{ψ}. The *path formula* φ represents (the disjunction of) all paths from l^{ψ} to the abstract state e. The top element is $\top = (true, l_\top, true)$. The partial order $\sqsubseteq \subseteq E \times E$ is defined for any two abstract states $e_1 = (\psi_1, l^{\psi_1}, \varphi_1)$ and $e_2 = (\psi_2, l^{\psi_2}, \varphi_2)$ as $e_1 \sqsubseteq e_2 \Leftrightarrow (e2 = \top) \vee ((l^{\psi_1} = l^{\psi_2}) \wedge (\psi_1 \wedge \varphi_1 \Rightarrow \psi_2 \wedge \varphi_2))$. The join operator $\sqcup : E \times E \to E$ is based on the partial order and returns the least upper bound of its operands.
(2) The transfer relation $\rightsquigarrow_{\mathbb{P}}$ has the transfer $e \overset{g}{\rightsquigarrow}_{\mathbb{P}} e'$ for an edge $g = (l, op, l')$ and two abstract states $e = (\psi, l^{\psi}, \varphi)$ and $e' = (\psi', l^{\psi'}, \varphi')$, if $(\psi', l^{\psi'}, \varphi') = \begin{cases} (true, l', (SP_{op}(\varphi) \wedge \psi)^{\Pi}) & \text{if } blk(e, g) \\ (SP_{op}(\varphi), l', \psi) & \text{otherwise} \end{cases}$, where $SP_{op}(\varphi)$ denoted the strongest post of a given path formula φ for an operation op. The choice of computing a boolean predicate abstraction depends on the configurable operator blk. For our work it returns *true* at least for procedure calls, procedure entries, and procedure exits. The boolean predicate abstraction $(\cdot)^{\Pi}$ computes the strongest boolean combination of predicates $\mathcal{P}$.
(3) The merge operator $\text{merge}_{\mathbb{P}} : E \times E \to E$ combines the two abstract states $e_1 = (\psi_1, l^{\psi_1}, \varphi_1)$ and $e_2 = (\psi_2, l^{\psi_2}, \varphi_2)$ according to their last abstraction computation: $\text{merge}(e_1, e_2) = \begin{cases} (\psi_2, l^{\psi_2}, \varphi_1 \vee \varphi_2) & \text{if } (\psi_1 = \psi_2) \wedge (l^{\psi_1} = l^{\psi_2}) \\ e_2 & \text{otherwise} \end{cases}$
(4) The termination check $\text{stop}_{\mathbb{P}} = \text{stop}_{sep}$ returns whether a covering abstract state was already reached before.
(5) For an abstract state $e_I = (\psi_I, l^{\psi_I}, true)$ at a block entry, the operator $\text{reduce}_{\mathbb{P}}$ computes the set $\mathcal{P}_{\neg B} := \{p_1, ..., p_i\}$ of predicates that are irrelevant for the block abstraction and removes them from the abstraction formula: $\text{reduce}_{\mathbb{P}}(B, e_I) = ((\exists p_1, ..., p_i : \psi_I), l^{\psi_I}, true)$.
(6) The operator $\text{expand}_{\mathbb{P}}$ contradicts the operator $\text{reduce}_{\mathbb{P}}$, computes the set $\mathcal{P}_B := \{p_{i+1}, ..., p_n\}$ of predicates, and restores the full set of predicates $\mathcal{P} = \mathcal{P}_{\neg B} \uplus \mathcal{P}_B$ for an output state $e_o = (\psi_o, l^{\psi_o}, true)$. Therefore the abstraction formula ψ_o is extended with the remaining part of the initial abstraction formula ψ_I: $\text{expand}_{\mathbb{P}}(e_I, B, e_o) = ((\exists p_{i+1}, ..., p_n : \psi_I) \wedge \psi_o, l^{\psi_o}, true)$.
(7) The operator $\text{rebuild}_{\mathbb{P}}$ is based on the procedure-call state $e_{call} = (\psi_{call}, l^{\psi_{call}}, true)$, the (not reduced) procedure-entry state $e_I = (\psi_I, l^{\psi_I}, true)$, and the expanded procedure-exit state $e_O = (\psi_O, l^{\psi_O}, true)$. The path formula φ_{call} represents the CFA edge that is the procedure entry edge between the program locations of the abstract states e_{call} and e_I and represents the encoding of all assignments of the arguments to the parameter variables. The operator $\text{rebuild}_{\mathbb{P}}$ is defined as $\text{rebuild}_{\mathbb{P}}(B, e_{call}, e_I, e_O) = (\psi_{call} \wedge \varphi_{call} \wedge \psi_O)^{\Pi}$ and computes the predicate abstraction for the conjunction of the abstractions before and after the procedure call and the parameter assignment.

Interval-CPA. The CPA for *interval analysis* $\mathbb{I} = (D_{\mathbb{I}}, \rightsquigarrow_{\mathbb{I}}, \text{merge}_{\mathbb{I}}, \text{stop}_{\mathbb{I}})$ tracks variables and the range (interval) of their possible assigned values. The *interval analysis* is similar to the *value analysis*, which can be seen as a special case using intervals containing only one value. The coverage relation between intervals is based on the inclusion of intervals (instead of

equality of values). We omit the detailed definition here to keep the reader focused on our approach.

4.1 Soundness of Reduce and Expand Operator for the Given Domains

For each of the described domains, the soundness criterion of the whole interprocedural analysis is based on the soundness of the CPA algorithm itself (which we assume as basis) as well as on the properties of the specific operators reduce and expand. For a sound analysis, the abstract states that would have been reached without applying a block abstraction (i. e., only applying the wrapped CPA $\mathbb{W}$) need to be a subset of the states reached with an application of the corresponding block abstraction, i. e., using block abstractions can only be less precise than a default analysis, but never cut off a reachable part of the abstract state space.

The transfer relation $\rightsquigarrow_{\mathbb{BAM}}$ for an abstract state $e \in E$ satisfies the relation $\{e' \in E \mid e \rightsquigarrow_{\mathbb{W}} e'\} \subseteq \{e'' \in E \mid e \rightsquigarrow_{\mathbb{BAM}} e''\}$ Based on the previous definition of $\rightsquigarrow_{\mathbb{BAM}}$ the interesting case appears when applying a block abstraction. Thus, the concrete implementation of the operators reduce and expand must satisfy the following condition: $\{e' \in E \mid e \rightsquigarrow_{\mathbb{W}} e'\} \subseteq \{\mathsf{expand}(e, B, e_o) \in E \mid \mathsf{reduce}(B, e) \rightsquigarrow_{\mathbb{W}} e_o\}$.

For the call-stack analysis, each abstract call-stack state after an application of a block abstraction exactly matches the call-stack state without such a block abstraction. To proof this, just extend each call stack during the block analysis with the removed part $[f_1, ..., f_{i-1}]$ from the reduce operation. For the value analysis (and based on a programming language without pointer handling), the same proof can be applied: Removing assignments from abstract states and restoring them later results in an abstract state that matches the state when not applying a block abstraction computation. A detailed soundness proof for the predicate domain is given in [49]. Removing irrelevant predicates $P_{\neg B}$ and conjuncting those predicates when applying the block abstraction does only make the analysis more imprecise, but does not limit the reachable abstract state space.

4.2 Embedding BAM Interprocedural in CEGAR

The framework CPAchecker defines BAM as a CPA and allows to combine the CPA algorithm with other reachability algorithms like CEGAR [23] that allows to refine the granularity of the abstract analysis by information extracted from infeasible counterexamples. Additional operators for the refinement step in CEGAR are also defined in domain-independent manner and available in the framework. In our case, the CEGAR algorithm can wrap the CPA algorithm and the analysis of BAM can benefit from this. Whenever BAM finds a property violation, the reachability analysis and the fixed-point algorithm terminates and the surrounding CEGAR algorithm checks the error path for feasibility. If necessary, CEGAR refines the precision, and BAM with the fixed-point algorithm is re-started with the updated precision.

In case of the *predicate analysis*, the refinement procedure computes tree interpolants [17, 33] according to procedure scopes, i.e., for each entered (and exited) procedure scope along a (spurious) counterexample trace, a new subtree for the tree interpolation problem is constructed. For other analysis, like *value analysis*, the refinement of recursive procedures does not need special handling. In this case, a refinement strategy for sequential counterexamples [15] is sufficient.

4.3 Detailed Description of the Example

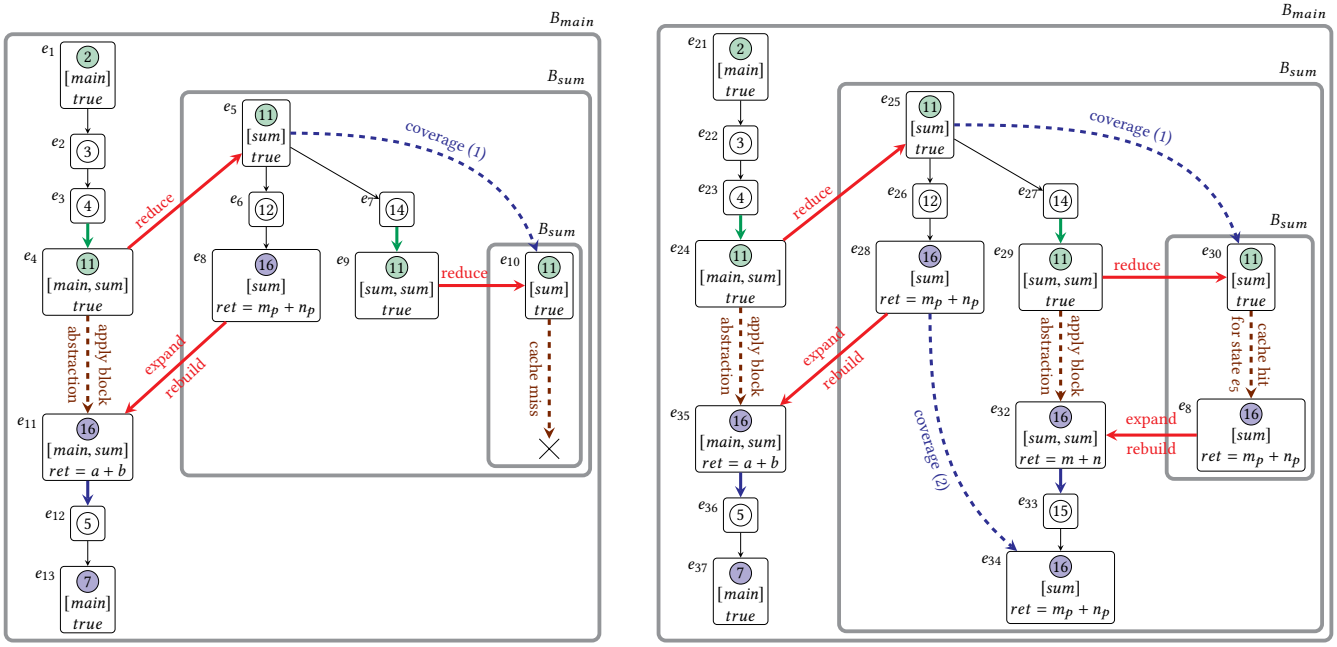
In the following, we provide deeper insights for the previously given example program (see Sect. 1) in Fig. 1 to show the control flow of BAM with the fixed-point algorithm when using the *predicate analysis*. We combine the previously defined Callstack-CPA $\mathbb{C}$ and the Predicate-CPA $\mathbb{P}$, i. e., the transfer relation, coverage check, and reduce, expand, or rebuild operators are applied in both domains.

Figure 5 shows the abstract states that are reached in the first two iterations of the fixed-point algorithm, which terminates after the second iteration. The labeling of each abstract state consists of the program location (circled number in first line), the call stack (second line), and the abstraction formula of the *predicate analysis* (third line). To keep the figure readable, we dismiss the call stack and abstraction formula whenever there is no change in the abstract state. In the upper left corner, each node is annotated with an index that refers to the exploration strategy and control flow of the analysis.

The operators reduce, expand, and rebuild show their effect at the program locations 11 and 16, which are the input and output locations of the procedure blocks B_{main} and B_{sum}. For example, the operator $\mathsf{reduce}_{\mathbb{C}}$ of the *call-stack analysis* removes of all procedure symbols except the most local one from the call stack. The operator $\mathsf{expand}_{\mathbb{C}}$ restores the whole call stack when the analysis leaves the block. The effect of the $\mathsf{rebuild}_{\mathbb{P}}$ at program location 16 will be described below.

Initialization. We assume that the initial cache and the stack of BAM are empty and the following set of predicates is defined as precision: $\mathcal{P} := \{ret = m_p + n_p, ret = a + b, m = m_p \wedge n = n_p\}$. The *predicate analysis* uses the symbols m_p, n_p, and ret to encode parameter assignments at function entry and the return value. Such predicates can be generated via an interpolation procedure from previously found spurious counterexample traces in the context of CEGAR. For simple programs (like this example) they match the expected procedure summary. In general, the analysis might need several iterations of CEGAR to receive a sufficiently good precision. In this example, we concentrate on the rebuild operator. All predicates are relevant for the block B_{sum}, i. e., $\mathcal{P}_B = \mathcal{P}$, i.e., the reduce and expand operators for *predicate analysis* will keep the abstraction formula unchanged.

First iteration. The result of the first iteration of the fixed-point loop is shown in Fig. 5a. The analysis starts with the initial abstract state e_1 at the program location 2, entering the main block B_{main} and pushing e_1 onto the stack. The recursive procedure block B_{sum} is analyzed for the first time at the procedure call from program location 4 to program location 11, where BAM starts a new subanalysis with state e_4 for the block B_{sum}. The reduction removes the suffix *main* of the call stack and keeps the abstraction formula $true$. The abstract state e_5 is added into the stack. When the procedure block B_{sum} is entered the second time (procedure call at program location 14 for state e_9), the reduced abstract state e_{10} is compared



(a) After first iteration; cache miss leads to second iteration

(b) After second iteration; fixed point is reached

Figure 5: Graph of reached abstract states after the first two fixed-point iterations

with elements from the stack. The coverage relation (Alg. 2, line 2) is satisfied. BAM has no computed procedure summary in the cache and returns an empty set of reachable abstract states (line 6 of Alg. 2). The flag fixedpointReached is set to *false* in line 7 of Alg. 2. The analysis continues with the exploration of the non-recursive branch of the procedure. When leaving block B_{sum}, the block's summary is inserted into the cache, i. e., the block abstraction from the abstract state e_5 towards the abstract state e_8 is stored for later usage. For the predicate analysis, the summary of the block is the abstraction formula $ret = m_p + n_p$, which describes the dependency between the function parameters and the return value.

The rebuild operator $\text{rebuild}(B, e_3, e_4, e_8)$ restores information from the calling context. Using the abstraction formula $\psi_3 := true$, the parameter assignment from the procedure call $\varphi_{call} := (a = n_p \wedge b = m_p)$, and the block summary $\phi_8 := (ret = m_p + n_p)$, the rebuild operator $\text{rebuild}_{\mathbb{P}}$ computes $(\psi_3 \wedge \varphi_{call} \wedge \psi_8)^{\Pi} = (ret = a + b)$, i. e., based on the given predicates, the procedure is summarized with $ret = a + b$, which describes the dependency between the function arguments and the return value. We do not describe internals of *predicate abstraction* here, but refer to the literature [14]. No property violation is found along the path until state e_{13}, i. e., the branching towards program location 6 is not satisfiable, and the fixed-point loop continues.

Second iteration. The initial steps of this iteration are similar to the first iteration. After a few steps, the stack consists of the abstract states e_{21} and e_{25}. A different control flow appears when the analysis reaches the recursive procedure call again at state e_{30}, with a coverage relation for the abstract state e_{25} that is part of the stack. Now we get a cache hit for the previously computed block abstraction between state e_5 and state e_8 and apply the procedure summary to skip the recursive procedure call (line 4 of Alg. 2). Using the abstraction formula $\psi_{27} := true$, the parameter assignment from the procedure call $\varphi_{call} := (n = n_p \wedge m = m_p)$, and the block summary $\phi_8 := (ret = m_p + n_p)$, the rebuild operator $\text{rebuild}_{\mathbb{P}}$ computes $(\psi_{27} \wedge \varphi_{call} \wedge \psi_8)^{\Pi} = (ret = m + n)$. When leaving the procedure blocks, our approach (Alg. 2, line 19) checks for new (not yet covered) abstract states. In this example, state e_{34} is already covered by state e_{28}, thus the fixed-point algorithm terminates after this iteration. As the property violation at program location 6 is not reachable, the program is verified.

5 EXPERIMENTAL EVALUATION

We evaluate BAM Interprocedural for several domains and show that it is competitive with existing approaches. We structure the evaluation according to two claims:

Claim I: Domain-Independence and Modularity. We claim that our interprocedural approach is domain-independent and can be implemented in a modular way. To evaluate the claim, we apply the approach to several abstract domains, show that the analysis works, and make the implementation publicly available.

Claim II: Effectiveness and Efficiency. We claim that our approach —despite the modular design— does not cause large overheads in the analysis. To evaluate the claim, we have to compare the effectivity (number of solved problems) and performance (runtime) of our implementation against state-of-the-art verification tools.

5.1 Benchmark Programs and Setup

We use all 104 programs from the SV-COMP'20 benchmark set [1] on recursion. Most programs are generic and allow to easily scale the programs for deeper recursion; they include recursive algorithms, e.g., Fibonacci, Ackermann, Towers of Hanoi, and McCarthy91.

All experiments were performed on machines with a 3.4 GHz Quad Core CPU and 33 GB of RAM. The operating system was Ubuntu 18.04 (64 bit) with Linux 4.15.0. A CPU time limit of 15 min and a memory limit of 15 GB were used, which seems to be standard. Measurements and resource limits were managed by BenchExec [16].

5.2 Results and Discussion

Claim I. We implemented our domain-independent approach in CPAchecker for several domains, including *value analysis*, *predicate analysis*, and *interval analysis*. In addition, we evaluated a reduced product [27] of value and predicate analysis. We used the CPAchecker version that participated in SV-COMP'20. CPAchecker was chosen as the implementation platform because it has a configurable and modular design which is easy to extend by new concepts, has a large user base, and is well maintained.[2]

Table 1 compares BAM Interprocedural for four domains (one of them being a product), by providing the CPU time (in seconds, with three significant digits) needed by the verifiers for all correctly solved verification tasks and the number of correctly solved tasks, divided into proofs and bugs found. In sum, the new approach supports the interprocedural analysis of recursive procedures for all three domains separately as well as for a combination of domains.

Claim II. We provide the results of state-of-the-art software verifiers, which participated in SV-COMP'20.[3] We compared 13 verifiers that participated successfully in the category "Recursive" of SV-COMP. This includes the predicate-based verifiers CPAchecker [30, 48], PeSCo [29, 43] and Ultimate Automizer [32, 34], the bounded model checkers Cbmc [25, 36] and Esbmc [31, 39], the symbolic-execution tool Symbiotic [20, 21], as well as the SMT-based tool Map2Check [44, 45]. The binary archives of all verifiers are publicly available.[4] The data are extracted from the published SV-COMP'20 results [7].

Table 2 provides the sum of CPU time needed by the verifiers for all correctly solved verification tasks, and the number of correctly solved tasks, divided into proofs and bugs found. The configuration used by verifier CPAchecker (SV-COMP'20) combines *value analysis* and *predicate analysis* within our interprocedural approach (same configuration as in the last row of Table 1), which is selected from the configuration as the strategy to verify recursive programs [8]. The performance of the tool with our approach (CPAchecker) shows that although modular and domain-independent, it is competitive in terms of *effectiveness* and *efficiency*: BAM Interprocedural solves about as many tasks as the other tools within reasonable CPU time. None of the tools managed to verify all tasks; there are several tasks in the given benchmark set that could not be solved by any verifier.

[1] https://github.com/sosy-lab/sv-benchmarks/tree/svcomp20
[2] https://www.openhub.net/p/cpachecker
[3] https://sv-comp.sosy-lab.org/2020/
[4] https://gitlab.com/sosy-lab/sv-comp/archives-2020/-/tree/svcomp20

Domain	CPU time (s)	Proofs	Bugs
Value	955	31	37
Predicate	2 600	28	37
Interval	858	36	38
Value + Predicate	2 130	37	46

Table 1: Results for the comparison of different abstract domains with BAM Intraprocedural in CPAchecker

Verifier	CPU time (s)	Proofs	Bugs
Cbmc	662	32	47
CPAchecker (SV-COMP'20)	2 180	37	46
Divine	1 190	32	42
Esbmc	941	33	47
Map2Check	23 600	34	37
PeSCo	3 130	37	46
Pinaka	237	31	31
Symbiotic	138	33	45
Ultimate Automizer	2 160	41	37
Ultimate Kojak	1 010	19	28
Ultimate Taipan	6 210	42	37
VeriAbs	7 630	41	46
VeriFuzz	1 960	0	45

Table 2: Results for the comparison of different verifiers

6 CONCLUSION

We have presented BAM Interprocedural, a novel approach to interprocedural program analysis. The new approach is **modular** and **domain-independent**, because it is not integrated in a specific program analysis but wraps an existing analysis. In other words, given an arbitrary abstract domain for *intra*-procedural data-flow analysis, we can turn it into an *inter*-procedural analysis without much (a) development work and (b) performance overhead. We have illustrated in detail how to make *predicate analysis* and *value analysis* interprocedural. Our implementation and experiments show that BAM Interprocedural works well for four different program analyses. The new approach supports **recursive** procedures, because it is not bounded to procedure scopes. We showed the effectiveness on the benchmark set of recursive programs from SV-COMP'20: the approach is able to successfully verify recursive procedures. The new approach is **efficient**, because it is integrated into BAM and does not add much overhead on top of the wrapped abstract domain. Compared to other software verifiers, the new implementation is competitive. Due to the modular approach, the effectiveness and efficiency heavily depends on the wrapped program analysis. Our results are promising and there is potential for optimization in our implementation, which we will explore in future work. We plan to specify the operator rebuild for further domains like binary decision diagrams, symbolic memory graphs, or octagons, e.g., to analyze more difficult memory-accesses in recursive programs.

We hope that other researchers and developers of verification tools can benefit from our approach because it separates the concern of making an analysis interprocedural from the actual work on implementing and improving abstract domains.

REFERENCES

[1] 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

[2] Anonymous. 2020. Replication Package for Article 'Domain-Independent Interprocedural Program Analysis using Block-Abstraction Memoization. Zenodo. https://doi.org/10.5281/zenodo.3698236

[3] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Proc. IFM (LNCS 2999)*. Springer, 1–20. https://doi.org/10.1007/978-3-540-24756-2_1

[4] 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

[5] 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

[6] D. Beyer. 2020. Advances in Automatic Software Verification: SV-COMP 2020. In *Proc. TACAS (2) (LNCS 12079)*. Springer. https://www.sosy-lab.org/research/pub/2020-TACAS.Advances_in_Automatic_Software_Verification_SV-COMP_2020.pdf

[7] D. Beyer. 2020. Results of the 9th International Competition on Software Verification (SV-COMP 2020). Zenodo. https://doi.org/10.5281/zenodo.3630205

[8] 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

[9] 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

[10] 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

[11] D. Beyer and K. Friedberger. 2018. In-Place vs. Copy-on-Write CEGAR Refinement for Block Summarization with Caching. In *Proc. ISoLA (LNCS 11245)*. Springer, 197–215. https://doi.org/10.1007/978-3-030-03421-4_14

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] R. Blanc, A. Gupta, L. Kovács, and B. Kragl. 2013. Tree Interpolation in Vampire. In *Proc. LPAR (LNCS 8312)*. Springer, 173–181. https://doi.org/10.1007/978-3-642-45221-5_13

[18] 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

[19] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[20] M. Chalupa, J. Strejcek, and M. Vitovská. 2018. Joint Forces for Memory Safety Checking. In *Proc. SPIN*. Springer, 115–132. https://doi.org/10.1007/978-3-319-94111-0_7

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

[22] Y. -F. Chen, C. Hsieh, M. -H. Tsai, B. -Y. Wang, and F. Wang. 2014. Verifying Recursive Programs Using Intraprocedural Analyzers. In *Proc. SAS (LNCS 8723)*. Springer, 118–133. https://doi.org/10.1007/978-3-319-10936-7_8

[23] 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

[24] 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

[25] E. M. Clarke, D. Kröning, and F. Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proc. TACAS (LNCS 2988)*. Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[26] 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

[27] P. Cousot and R. Cousot. 1979. Systematic design of program-analysis frameworks. In *Proc. POPL*. ACM, 269–282. https://doi.org/10.1145/567752.567778

[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] M. Czech, E. Hüllermeier, M.-C. Jakobs, and H. Wehrheim. 2017. Predicting Rankings of Software Verification Tools. In *Proc. SWAN*. ACM, 23–26. https://doi.org/10.1145/3121257.3121262

[30] M. Dangl, S. Löwe, and P. Wendler. 2015. CPACHECKER with Support for Recursive Programs and Floating-Point Arithmetic (Competition Contribution). In *Proc. TACAS (LNCS 9035)*. Springer, 423–425. https://doi.org/10.1007/978-3-662-46681-0_34

[31] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole. 2018. ESBMC 5.0: An Industrial-Strength C Model Checker. In *Proc. ASE*. ACM, 888–891. https://doi.org/10.1145/3238147.3240481

[32] M. Heizmann, D. Dietsch, J. Leike, B. Musa, and A. Podelski. 2015. ULTIMATE AUTOMIZER with Array Interpolation. In *Proc. TACAS (LNCS 9035)*. Springer, 455–457. https://doi.org/10.1007/978-3-662-46681-0_43

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

[35] 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

[36] D. Kröning and M. Tautschnig. 2014. CBMC: C Bounded Model Checker (Competition Contribution). In *Proc. TACAS (LNCS 8413)*. Springer, 389–391. https://doi.org/10.1007/978-3-642-54862-8_26

[37] 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

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

[39] J. Morse, M. Ramalho, L. C. Cordeiro, D. Nicole, and B. Fischer. 2014. ESBMC 1.22 (Competition Contribution). In *Proc. TACAS (LNCS 8413)*. Springer, 405–407. https://doi.org/10.1007/978-3-642-54862-8_31

[40] P. Müller and T. Vojnar. 2014. CPALIEN: Shape Analyzer for CPACHECKER (Competition Contribution). In *Proc. TACAS (LNCS 8413)*. Springer, 395–397.

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

[42] 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

[43] C. Richter and H. Wehrheim. 2019. PESCO: Predicting Sequential Combinations of Verifiers (Competition Contribution). In *Proc. TACAS (3) (LNCS 11429)*. Springer, 229–233. https://doi.org/10.1007/978-3-030-17502-3_19

[44] H. Rocha, R. S. Barreto, and L. C. Cordeiro. 2015. Memory Management Test-Case Generation of C Programs Using Bounded Model Checking. In *Proc. SEFM (LNCS 9276)*. Springer, 251–267. https://doi.org/10.1007/978-3-319-22969-0_18

[45] H. O. Rocha, R. Barreto, and L. C. Cordeiro. 2016. Hunting Memory Bugs in C Programs with Map2Check (Competition Contribution). In *Proc. TACAS (LNCS 9636)*. Springer, 934–937. https://doi.org/10.1007/978-3-662-49674-9_64

[46] O. Sery, G. Fedyukovich, and N. Sharygina. 2011. Interpolation-Based Function Summaries in Bounded Model Checking. In *Proc. HVC (LNCS 7261)*. Springer, 160–175. https://doi.org/10.1007/978-3-642-34188-5_15

[47] O. Sery, G. Fedyukovich, and N. Sharygina. 2015. Function Summarization-Based Bounded Model Checking. In *Validation of Evolving Software*. Springer, 37–53. https://doi.org/10.1007/978-3-319-10623-6_5

[48] D. Wonisch. 2012. Block Abstraction Memoization for CPACHECKER (Competition Contribution). In *Proc. TACAS (LNCS 7214)*. Springer, 531–533. https://doi.org/10.1007/978-3-642-28756-5_41

[49] 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