Proc. ISoLA 2018, © Springer

In-Place vs. Copy-on-Write CEGAR Refinement for Block Summarization with Caching

Dirk Beyer and Karlheinz Friedberger

LMU Munich, Germany

Abstract. Block summarization is an efficient technique in software verification to decompose a verification problem into separate tasks and to avoid repeated exploration of reusable parts of a program. In order to benefit from abstraction at the same time, block summarization can be combined with counterexample-guided abstraction refinement (CEGAR). This causes the following problem: whenever CEGAR instructs the model checker to refine the abstraction along a path, several block summaries are affected and need to be updated. There exist two different refinement strategies: a destructive *in-place* approach that modifies the existing block abstractions and a constructive *copy-on-write* approach that does not change existing data. While the *in-place* approach is used in the field for several years, our new approach of *copy-on-write* refinement has the following important advantage: A complete exportable proof of the program is available after the analysis has finished. Due to the benefit from avoiding recomputations of missing information as necessary for *in-place* updates, the new approach causes almost no computational overhead overall. We perform a large experimental evaluation to compare the new approach with the previous one to show that full proofs can be achieved without overhead.

Keywords: Software Model Checking, Block Summarization, Copy-on-Write, CEGAR, Abstraction Refinement, CPAchecker, Program Analysis

1 Introduction

Software model checking is a powerful technique for proving programs correct as well as for finding errors in programs. Given a program and a specification, a model checker either finds an error path through the program that exposes the specification violation or proves that the specification is satisfied by the program. In this paper, we take a look at the combination of two orthogonal approaches, *block summaries* and *abstraction refinement*.

The technique of constructing summaries of program blocks [18] is effective to reduce the overhead that an exploration without summaries would otherwise cause. Block-abstraction memoization (BAM) [27] is based on a standard state-space exploration using a given control-flow automaton (CFA) that represents the program. The CFA is partitioned into blocks, which are analyzed separately by BAM. Block abstractions (e.g., the results of a block's analysis) represent summaries of

blocks. Block abstraction is a generalization of function summaries, if the block size is chosen according to function bodies. In general, block abstraction also works for loop bodies and other block definitions. Block abstractions are stored in a cache, such that they can be reused whenever the same block is explored again. The exact behavior of the analysis and the precision of BAM is determined by a wrapped underlying analysis, such as predicate analysis or explicit-value analysis.

Abstraction, i.e., verifying an overapproximating abstract model of the program instead of its concrete state space, is an idea for scaling model checking to large programs orthogonal to summaries. The verification of the abstract model is often less complex and more resource-efficient. Counterexample-guided abstraction refinement (CEGAR) [15] is a property-directed approach for the automatic construction of an abstract model for a given system: it automatically determines a level of abstraction for program verification that is coarse enough to omit unnecessary information from the abstract model and precise enough to refute spurious counterexamples. The basic idea is to iteratively identify relevant facts from infeasible program paths and use them for the further and more precise state-space exploration. Many existing software model-checking algorithms are based on this approach, such as predicate analysis [8], IMPACT [23], and explicit-value analysis [12].

Our combination of block abstraction with CEGAR needs a special refinement strategy such that only the necessary parts of the (cached) state space are touched. However, block abstractions are cached and can be used at different locations during the analysis and even several times on the same error path. The problem is how to correctly refine the block abstractions in the context of BAM based on a the underlying refinement strategy. The original definition of refinement in BAM [27] describes a destructive *in-place* update of block abstractions and explains that *holes* occur in the state space which are caused by modifications on existing block abstractions. Those holes need to be recomputed on demand. However, this is not possible in general, e.g., after the analysis has finished, because the information which block abstraction was computed for which block is no longer accessible. Succeeding analysis steps are not able to recompute the missing information, as not only the block abstractions themselves, but also their dependencies are deleted in the destructive approach. A recomputation would imply to rerun a large part of the complete analysis. Due to the unforeseeable appearance of cache accesses, the recomputation might even produce a completely different counterexample or proof than the previous analysis.

The user usually wants the model checker to terminate with a proof, which in this setting might be an abstract reachability graph (ARG). The ARG is expected to include all initial abstract states and all abstract states that are reachable from the initial abstract states. This guarantee does not hold if the ARG contains holes. Succeeding analysis steps that are executed after the termination of the block-abstraction-based analysis and depend on the full abstract state space (without holes) have no possibility to recompute the missing parts. For example, correctness witnesses [6, 22] can not be reliably produced with block-abstraction-based analyses [1]: the exported correctness witness is either invalid because no graph from root to all reached abstract states could be

written, or a missing part in the correctness witness (branch in the graph) is responsible for incorrectly guiding the witness validator.

The main contribution of this paper is a new refinement approach based on a constructive *copy-on-write* strategy. Our work includes a comparative evaluation of the new *copy-on-write* approach with the previous *in-place* refinement, showing that the new approach has only a small computational overhead for run time and memory usage. Because BAM is independent of (and orthogonal to) other analyses in a full program analysis, it can be combined with analyses based on different abstract domains like predicate, value, or interval analysis [11, 12], or combinations thereof [1, 17]. Our new refinement strategy is fully integrated into BAM in CPAchecker and does not depend on the underlying analysis. Thus, there is no change in the behavior of the sub-analyses.

Contributions. We make the following contributions:

- We design a *copy-on-write* approach that solves two open problems: (i) strictly monotonic refinement for summary-based approaches in combination with CEGAR and (ii) abstract reachability graphs without holes that cause problems in later steps of the analysis.
- We implement the approach of *copy-on-write* refinement in the verification framework CPAchecker and make the source code available to others.[1]
- We experimentally evaluate the new approach on a large number of verification tasks to show that the *copy-on-write* approach is about as efficient and effective as the *in-place* approach, although the approach produces *complete* abstract reachability graphs.
- We make all experimental results, including raw data, tables, experiment setup, etc., available on a supplementary web site.[2]

Related Work. There are several techniques based on block-based summarization, as this idea dates back to Hoare [20]. The special case of *function summaries* aims at scalability for interprocedural analyses and is integrated in several algorithms and tools.

FunFrog [25, 26] uses an SMT solver and Craig interpolation to compute function summaries in the context of bounded model checking. Starting from an initially empty set of function summaries, the tool explores the problem's traces and computes interpolants from path formulas for all missing procedure calls. The interpolants are then directly used as summaries. This strategy is applied in a CEGAR loop until the specified property can be proven or is definitely violated. FunFrog uses a cache for function summaries, but does never modify existing function summaries.

Bebop [3] and Saturn [28] use binary decision diagrams (BDDs) and SMT to encode the program's semantics. The function summary is build by renaming variables in formulas, such that the direct encoding of a procedure call can be

[1] `https://cpachecker.sosy-lab.org`
[2] `https://www.sosy-lab.org/research/bam-cow-refinement`

reused several times within the same encoding of the program behavior. Both tools work on a very precise abstraction level and do not refine their summarizations.

BAM is a domain-independent approach for caching and reusing block abstractions. It is independent of functions and can be used with an arbitrary block size. Instead of being limited to a special domain like BDDs, SMT, or intervals, BAM works on an abstract level and can be applied to any abstract domain or even combinations of several domains, including predicate analysis and explicit value analysis [1]. The integration of CEGAR refinement in BAM was already described in the context of predicate analysis [27]. Our new approach of *copy-on-write* refinement for BAM makes the approach really lazy (matching the principles of *lazy abstraction refinement* [19]).

2 Background on Block Summarization

The following section provides an overview of some basic concepts and definitions that we use for our approach. We describe the program representation and the most important details of block-abstraction memoization that are used for state-space exploration (cf. other literature on block-abstraction memoization for more detailed descriptions [2, 7, 27]).

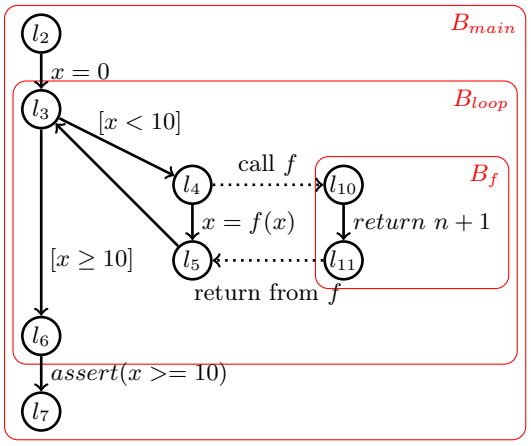
2.1 Program and State-Space Representation

A program is represented by a *control-flow automaton* (CFA) $A = (L, l_0, G)$, which consists of a set L of program locations (modeling the program counter), a set $G \subseteq L \times Ops \times L$ (modeling the control flow), and an initial program location l_0 (entry point; initial call of the main function). The set Ops contains the operations of the program, i.e., assignment and assume operations, function calls, and function returns. Let V be the set of variables in the program. A *concrete data state* assigns a value to each variable from the set V; the set C contains all concrete data states. For every edge $g \in G$, the transition relation is defined by $\overset{g}{\rightarrow} \subseteq C \times \{g\} \times C$. If there exists a sequence of concrete data states $\langle c_0, c_1, ..., c_n \rangle$ with $\forall i \in [1, n] : \exists g : c_{i-1} \overset{g}{\rightarrow} c_i \wedge (l_{i-1}, g, l_i) \in G$, then state c_n is called *reachable* from c_0 for l_0, i.e., there exists a syntactic walk through the CFA.

We perform a reachability analysis that unrolls the program lazily [19] into an *abstract reachability graph* (ARG) [8]. An ARG $S = (N, E)$ is a directed acyclic graph, consisting of a set N of ARG nodes (representing the abstract program states, e.g., including program location and variable assignments) and a set $E \subseteq N \times N$ of edges modeling the transfer that leads from one abstract state to the next one. We define a *subgraph* $S_s = (s, N_s, E_s)$ as a connected component of an ARG $S = (N, E)$, starting at a given abstract state $s \in N_s$ (denoted as root), such that $N_s \subseteq N$, $E_s \subseteq E$, and $\forall s' \in N_s : (s', s'') \in E \Rightarrow (s'' \in N_s \wedge (s', s'') \in E_s)$.

```
1  void main() {
2      int x = 0;
3      while (x<10) {
4          x = f(x);
5      }
6      assert(x>=10);
7  }
8
9  int f(int n) {
10     return n+1;
11 }
```

(a) Source code



(b) CFA with blocks

Fig. 1: Example program and its control-flow automaton with 3 blocks

2.2 Block Summarization

Block-abstraction memoization (BAM) [27] is a generalization of several block-based summarization approaches [18, 24, 25]. BAM divides an input program into smaller parts, named *blocks*, to analyze them separately by summary construction. It uses an arbitrary block size and is not limited to function boundaries. In addition, BAM uses a cache to reuse block abstractions. The blocks allow us to abstract from the surrounding context, reducing computational overhead, and improving the performance of an analysis. The analysis of each block corresponds to an abstract initial state at the block-entry location and a set of abstract exit states at the block-exit locations (both described later). Block abstractions (e.g., the combination of initial states and exit states of a block's analysis) are stored in a cache, such that they can be reused whenever the same block is visited again.

Blocks. The basic components of BAM are *blocks*, which 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 connected program locations and a set $G' = \{(l_1, op, l_2) \in G \mid l_1, l_2 \in L'\}$ of control-flow edges. Two different blocks $B_1 = (L'_1, G'_1)$ and $B_2 = (L'_2, G'_2)$ are either disjoint ($L'_1 \cap L'_2 = \emptyset$) or one block is completely nested in the other block ($L'_1 \subset L'_2$). Each block $B = (L', G')$ has *entry* and *exit locations*, which are defined as $In(B) = \{l \in L' \mid (\exists (l', op, l) \in G \wedge l' \notin L') \vee \nexists (l', op, l) \in G\}$ and $Out(B) = \{l \in L' \mid (\exists (l, op, l') \in G \wedge l' \notin L') \vee \nexists (l, op, l') \in G\}$, respectively. In general, the block size can be freely chosen in BAM. In most cases, function and loop bodies are taken as blocks, because they represent logical structures of the program and seem to be a good choice for block abstraction.

Figure 1 shows the CFA (b) for an example program (a). The CFA is structured into three nested blocks B_{main}, B_{loop}, and B_f, such that their sizes align with

the function and loop bodies. In the example, each block has only one entry and one exit location, e.g., $In(B_{main}) = \{l_2\}$, $Out(B_{main}) = \{l_7\}$, $In(B_{loop}) = \{l_3\}$, $Out(B_{loop}) = \{l_6\}$, $In(B_f) = \{l_{10}\}$, and $Out(B_f) = \{l_{11}\}$.

State-Space Exploration with BAM. BAM is an algorithm for program analysis that makes use of a wrapped program analysis $\mathbb{W}$, which tracks data facts and does the *actual* (block-local) program-analysis work, i.e., computes abstractions, formulas for paths, or checking whether the property holds. Our framework is based on the concept of configurable program analysis (CPA) [9] and uses, for example, predicate analysis (based on SMT solving and predicates), explicit-value analysis (tracks assignments of variables), or combinations thereof, with usage of common basic components such as location analysis (tracks the program counter) or call-stack analysis (tracks the current call stack). Each CPA provides the analysis operators, like the transfer relation $\rightsquigarrow$ to compute abstract successor states for a specific abstract domain. BAM is specified as a CPA and does not know about the internals of the wrapped analysis $\mathbb{W}$, which is also a CPA. The approach of BAM just operates on abstract states of a (possibly combined) abstract domain to generate block abstractions.

The state-space exploration with BAM is defined recursively for blocks. The successor computation for abstract states chooses between two possible steps, depending on the currently analyzed program location: At an entry location of a block B, the successor computation $\overset{B}{\rightsquigarrow}_{\mathbb{B}}$ of the containing block executes a separate *nested sub-analysis* of the block B (starting with the initial abstract state for the block-entry location). This step produces a separate ARG that is later integrated as block abstraction into the surrounding analysis context. The block abstraction can either be computed or taken from a cache, if the block has been analyzed before. For block-exit locations of blocks, there is no succeeding abstract state (in the nested sub-analysis). For other program locations, the successor computation $\rightsquigarrow_{\mathbb{W}}$ is applied, which acts according to the abstract domain of the wrapped analysis $\mathbb{W}$ (e.g., tracks variables or computes abstractions for abstract states). Abstract states where a specification violation occurs are handled as if those abstract states are at block-exit locations of the current block, i.e., the nested sub-analysis terminates and returns the violating abstract states directly for the block abstraction.

Note that an ARG can contain edges representing block abstractions. The block of the block abstraction is inlined whenever a concrete program path without block abstractions is needed. This overhead is the necessary price for having a block-modular analysis. When CEGAR modifies the ARGs during the refinement, a problem occurs, which we will describe later.

2.3 CEGAR

Counterexample-guided abstraction refinement (CEGAR) is an approach to automatically adjust the granularity of an analysis by learning from infeasible error paths the relevant analysis facts that are needed to verify a program.

We use CEGAR as a wrapper algorithm around the state-space exploration algorithm, which is implemented as CPA algorithm [10]. The granularity of the analysis is defined as a *precision* that is refined in each iteration of the CEGAR algorithm. Each abstract state in an ARG has a precision. The precision of an abstract state can be changed during the refinement step. A too coarse precision would lead to an imprecise analysis that reports false alarms, a too fine precision would lead to an expensive state-space exploration; CEGAR tries to find the "right" level of abstraction in between.

CEGAR consists of two steps, an exploration step and a refinement step, which are executed alternatingly until a feasible error path is found (and a bug is reported) or all error paths are proven to be infeasible (and a proof can be reported): The exploration step computes new successor abstract states and builds the abstract state space in form of an ARG $G = (N, E)$, using the level of abstraction determined by CEGAR. When finding a possible specification violation, a feasibility check is applied, which examines the error path to the violation. A feasible error path is reported and the analysis terminates. An infeasible error path is used for a refinement step to gain more relevant facts from the program, e.g., by applying interpolation, and refine the precision. If the exploration step does not find any property violation and all abstract states are explored, the algorithm terminates and the program is proven correct.

The refinement step determines a *cut point* $s_{cut} \in N$ and a new precision for this position, such that the new level of abstraction is sufficient to exclude the infeasible error path from further exploration of the state space. The level of abstraction depends on the abstract domain of the analysis and might consist of, e.g., predicates (for predicate analysis) or a set of variables to be tracked (for value analysis). The outdated (too imprecise) subgraph $S_{s_{cut}} = (s_{cut}, N', E')$ of the already explored state space is removed from the ARG G and the subgraph's root state s_{cut} alone is re-added to G, such that the next exploration step of CEGAR recomputes this part of the state space with a higher precision. There are several approaches to determine the cut point s_{cut} along the error path [13]:

- **cut point at root:** full eager refinement is applied, where the whole explored state space is withdrawn and re-exploration starts from the initial root state of the ARG (e.g. [4, 14, 16]),
- **cut point as deep as possible:** only a (minimal) part of the explored state space is removed (lazy refinement [19]), such that a large part of the explored state space remains intact and can be reused in the further analysis (e.g. [8, 10]), or
- **cut point in between:** trade-off between reuse and reexploration is somewhere in between the above two choices [13].

The second approach performs best in most cases and is currently used in the field. As shown in Alg. 1, the refinement procedure first determines an abstract state s_{cut} where the infeasible subgraph is to be cut off and new facts are applied to the precision of the analysis. Lazy refinement is based on the idea that some parts of a program are analyzed with a coarse abstraction level and only some

Algorithm 1 Default refinement procedure of CEGAR

Input: an infeasible error path σ, an ARG G of the analysis
$s_{cut}, newFacts := \mathsf{refine}_{\mathbb{W}}(\sigma)$
$\mathsf{refinePrecision}(G, s_{cut}, newFacts)$
$\mathsf{removeSubgraph}(G, s_{cut})$

other parts of a program use a more fine-grained precision. Cutting off only a part of the ARG in each refinement fulfills this requirement.

2.4 Requirement for Refinement Approaches: New Precision Strictly More Precise

We use a (partial) order on precisions, such that a precision is considered as *more precise* compared to another precision, if it causes the analysis to track more information. For example, if an analysis uses a precision to track a set of variables or predicates (as predicate analysis and value analysis do), this relation is implicitly given by the subset relation. If a precision p is a superset of another precision p', then p is *more precise* than p'. Let an infeasible error path be a sequence of abstract states $\langle s_0, s_1, ...s_n\rangle$ with their precisions $\langle p_0, p_1, ...p_n\rangle$, such that s_0 is the root of the program and s_n violates the specification. The sequence $\langle p_0, p_1, ...p_n\rangle$ of precisions is *more precise* then a sequence $\langle p'_0, p'_1, ...p'_n\rangle$ of precisions if either p_0 is more precise than p'_0, or $p_0 = p'_0$ and the remaining sequence $\langle p_1, ...p_n\rangle$ of precisions is *more precise* than the sequence $\langle p'_1, ...p'_n\rangle$. We require a *refined* precision to be strictly more precise than its original, in order to guarantee *progress* in CEGAR (monotonic refinement).

Sine CEGAR is a fixed-point algorithm that starts with a coarse precision and refines it until it is sufficiently precise to prove or refute the program, the termination criterion for the CEGAR loop depends on a refinement approach that monotonically increases the precision. To ensure progress of the analysis, the refinement requirement needs to hold for each single refinement step in a program analysis with CEGAR.

Removing a subgraph $S_{s_{cut}} = (s_{cut}, N', E')$ from an ARG and applying a refined precision at its cut point s_{cut} fulfills the property, because the precision itself is more precise for the cut point, the predecessors are not touched, and the successors are deleted (and implicitly inherit the refined precision). Even if the removed subgraph $S_{s_{cut}}$ contained a more precise precision for some abstract state, the refinement requirement holds: Because the refined precision is represented as mapping from locations to precisions, and assigned as precision of the root abstract state s_{cut} of the subgraph, an ancestor of any removed state will be seeded with the new, refined precision. During re-exploration of the deleted subgraph, the analysis will re-explore prefixes of previously encountered error paths in this part of the state space and perform refinements of other error paths with cut points that also satisfy the refinement requirement. Strengthening the precision by additional information (like invariants from an external tool) before applying the update during the refinement also fulfills the property.

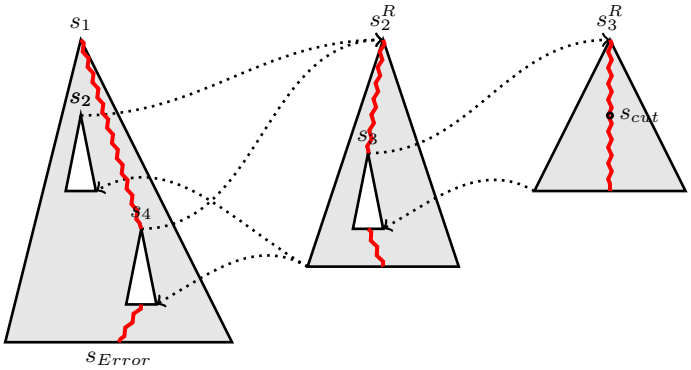


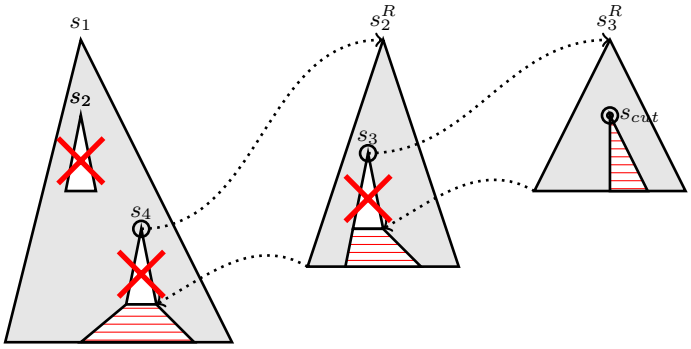
Fig. 2: State-space exploration with BAM and cut points for refinement

The refinement requirement is not fulfilled by the *in-place* approach, because the *in-place* refinement potentially deletes block abstractions for already analyzed parts of the state space and causes additional overhead if recomputation is needed for those missing block abstractions. The *copy-on-write* approach does not suffer from this problem.

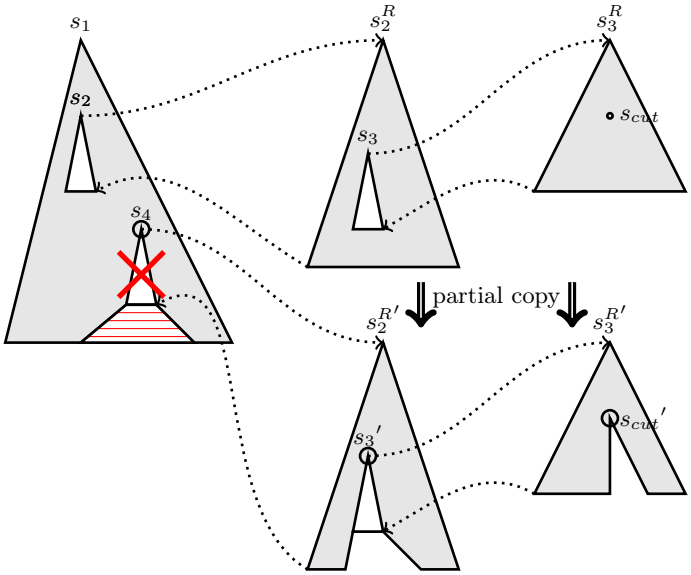
After defining all necessary parts, we will now go on with a motivating example, before giving the detailed description of the refinement approaches in BAM. Our goal is to replace an implementation that *in-place* modifies the ARG by a new *copy-on-write*-based approach for modifying the ARG. This allows us to efficiently keep the original as well as the copy for further processing. In the later evaluation we show that keeping the original data improves our analysis in several cases, and in particular, leaves the ARG complete (without holes).

3 Motivating Example

The following example illustrates the differences of the two strategies that could be used as refinement step in a CEGAR approach. In BAM, the analysis explores the state space and computes block abstractions for blocks. An example for such a state-space exploration is given in Fig. 2 (gray triangles represent ARGs, rooted at the top corner; white triangles represent block abstractions). We use block abstractions for nested blocks at the entry abstract states s_2, s_3, and s_4 with the corresponding initial abstract states s_2^R, s_3^R, and s_4^R in the nested analysis for the blocks at those program locations. In the example, let s_4^R be equal to s_2^R, such that we can reuse the existing block abstraction here. Block abstractions are shown as white triangles and are connected with their ARG via dotted lines. When finding the property violation s_{Error}, the analysis stops and performs a refinement for the found counterexample. The lazy refinement approach determines a possible cut-state, i.e., an abstract state s_{cut} along the error path, from where the found property violation is no longer reachable if a refined precision is applied.



(a) Strategy 1: In-place refinement step with removed block abstractions and subgraphs



(b) Strategy 2: Copy-on-write refinement step with copied ARGs and changes only in the most outer ARG

Fig. 3: In-place and copy-on-write refinement approach for BAM

At this point, the two refinement strategies differ:

Figure 3a shows the *in-place* refinement, removing parts of the explored state space, i.e., everything after the cut point s_{cut} and after the block abstractions (for abstract states s_3 and s_4). At the abstract state s_2 the *in-place* approach implicitly invalidates the ARG for the block abstraction and causes a *hole* in the surrounding ARG. Here, the applied block abstraction itself remains valid, because those abstract states were computed before the refinement.

Figure 3b shows the *copy-on-write* approach, updating the abstract states. All inner (nested) ARGs are updated *copy-on-write*. The red horizontal lines represent the removed abstract successor states after the block abstraction for abstract state s_4. The ARGs rooted at s_2^R and s_3^R are copied into new ARGs with roots at $s_2^{R'}$ and $s_3^{R'}$, leaving out the parts that are invalid after updating the precision. The references to or from block abstractions are also updated.

The difference in the refinement strategies is visible in Figs. 3a and 3b. While the first approach deletes and recomputes parts of the ARGs, the second approach works on fresh copies of the ARGs and uses them along with the old ARGs. In the following we explain both refinement strategies in more detail and discuss benefits of the second approach.

4 *In-Place* Refinement for BAM

The existing approach for refinement in BAM (as described earlier [27]) is sound, simple, and efficient, but has problems when abstract states need to be accessible afterwards. Briefly worded, the existing approach modifies cached block abstractions *in-place* and deletes important information that is not available after the refinement and needs to be recomputed for further steps of the analysis.

4.1 *In-Place* Refinement Algorithm for BAM

Algorithm 2 gives an overview of the *in-place* refinement of CEGAR for BAM, without going into detail for the further operation of BAM itself (cache management). The *in-place* refinement tries to mimic lazy abstraction refinement and CEGAR, i.e., it touches only a small number of abstract states and aims to update only those states where a precision update will avoid the re-exploration of the currently found infeasible error path. In contrast to Alg. 1, there is no single ARG G to work on, but with BAM there are several ARGs and the refinement must be applied to several of them. Algorithm 2 applies the following steps of the refinement:

After the refinement procedure of the underlying analysis has computed new facts for the analysis and determined an abstract state s_{cut} along the error path, the refinement approach determines the subgraph $S = (N, E)$ where the cut

Algorithm 2 In-place refinement procedure of CEGAR with BAM

Input: an infeasible error path σ
  $s_{cut}, newFacts := \mathsf{refine}_{\mathbb{W}}(\sigma)$
  $S := \mathsf{getARG}(s_{cut}, \sigma)$
  $\mathsf{refinePrecision}(S, s_{cut}, newFacts)$
  $\mathsf{removeSubgraph}(S, s_{cut})$
  while S is nested in another ARG S^* along σ **do**
    $s^* := \mathsf{getInitState}(S, S^*, \sigma)$
    $\mathsf{removeSubgraph}(S^*, s^*)$
    $S := S^*$

point s_{cut} is located. BAM might have used the block abstraction for S several times along the error path, and thus, we need to find out which outer subgraphs we need to remove (see Fig. 3a). Thus, we start from the correct block abstraction, and apply the removal operations for *in-place* refinement: The cut point s_{cut} and its subgraph $S_s(s_{cut}, N_s, E_s)$ get removed from S (with $s_{cut} \in N_s \subseteq N$ and $E_s \subseteq E$).

If the ARG S represents a block abstraction for a block B, i.e., S is nested within another ARG $S^* = (N^*, E^*)$, with the ARG S rooted at the initial abstract state $s^* \in N^*$, the subgraph S_{s^*} starting at the abstract state s^* of the nested-block abstraction is removed from the surrounding ARG S^*. This strategy is applied transitively up to the most outer ARG. The most outer block is not used as a block abstraction (it represents the whole program) and thus never referred to elsewhere in the state space.

The succeeding exploration step of CEGAR will re-explore the removed parts, use or recompute block abstractions and reach the abstract state with the refined precision, from where the state space is analyzed without exploring the previously encountered infeasible error path. Every ARG modification happens *in-place* and directly modifies the existing block abstractions. This approach does not consider whether a block abstraction was already used in another part of the state space, e.g., as part of another another ARG.

4.2 Problem of Cached Block Abstractions with *In-Place* Updates

The *in-place* refinement approach suffers from the *in-place* update of block abstractions in the following way: Whenever an missing abstract state belonging to a *hole* (missing block abstraction) in the state space is needed to be accessed, e.g., as part of a new error path, the block-entry state of the *hole*'s block abstraction is determined (depending on the context) and a possible valid block abstraction is recomputed. The previously updated ARG can not be used to fill the *hole*, because its precision might have been refined and updated *in-place*, such that it is more precise than before and leads to different block-exit abstract states. In order to not loose this refined precision for further exploration, all abstract states following the recomputed block abstraction need to be replaced by their recomputed counterparts (which also happens *in-place*).

In Fig. 3a this case happens when a property violation is found with an error path going through the removed block abstraction of s_2. Then the subgraph of s_2 needs to be removed and recomputed with a new block abstraction.

After BAM terminates (with or without finding a property violation) we often generate statistics or collect some data from the reached abstract state space. However, with *holes* there also comes the problem of missing data. This is only a minor problem, however might also irritate and mislead the user. Numeral statistics like the *number of abstract states* or the *number of predicates* are potentially misleading. Missing parts in non-numerical output, such as proofs and correctness witnesses, cause problems for later processing of the verification results. For example, we have identified several tasks in SV-COMP'17, for which CPAchecker (competition contribution BAM-BnB [1]) computed the correct

result during the analysis, but did not write a correctness witness for the validation, or a witness was written, but the graph of the witness was missing some parts, such that the witness validator was not correctly guided to some branches and could not successfully validate the result.

The new *copy-on-write* approach does not suffer from these problems, because the necessary data are kept until it is no longer needed, and we obtain correct statistics and valid (and complete) correctness witnesses.

5 *Copy-on-Write* Refinement for BAM

This section describes our new approach for *copy-on-write* refinement in BAM and considers the computational difference to *in-place* refinement.

5.1 *Copy-on-Write* Algorithm for BAM

We define a *copy* of an ARG $S = (N, E)$ as a second graph $S' = (N', E')$, where each ARG abstract state from N and transition from E is copied into N' and E'. Technically, a copy is just a new instance of the same ARG. Instead of changing an existing ARG S, whenever we would need to remove a subgraph S_s from it, the *copy-on-write* algorithm (Alg. 3) copies the ARG S into a new ARG S', omitting the corresponding subgraph. Then, we update the precision only for abstract states in the new instance S'. The new ARG S' is then registered in the cache as new block abstraction for one position where previously S was used, such that further explorations use the new instance S'. The old ARG S remains untouched, is still valid, and can be accessed when revisiting existing block abstractions.

When copying an ARG S that contains an embedded ARG S^{nested} from a nested sub-analysis, Alg. 3 only references the existing instance S^{nested} in the new ARG S' and does not copy it, except S^{nested} itself has to be modified. In this case, a copy $S^{nested\prime}$ is inserted instead of the original S^{nested}.

Algorithm 3 Copy-on-write refinement procedure of CEGAR with BAM

Input: an infeasible error path σ
  $s_{cut}, newFacts := \mathsf{refine}_{\mathbb{W}}(\sigma)$
  $S := \mathsf{getARG}(s_{cut}, \sigma)$
  $S', s'_{cut} := \mathsf{copyWithoutSubgraph}(S, s_{cut})$
  $\mathsf{registerARG}(S')$
  $\mathsf{refinePrecision}(S', s'_{cut}, newFacts)$
  while S' is nested in another ARG S^* along σ **do**
    $S^* := \mathsf{getInitState}(S', S*, \sigma)$
    $S^{*\prime}, s^{*\prime} := \mathsf{copyWithoutSubgraph}(S^{*\prime}, s^*)$
    $\mathsf{registerARG}(S^{*\prime})$
    $S' := S^{*\prime}$

Computational Overhead for *Copy-on-Write* Refinement. The run time of the *copy-on-write* refinement is similar to the *in-place* approach, because every affected ARG is exactly traversed once in each of the approaches. Thus, the run time of both refinement strategies is linear in the number of reached states. The conceptional difference comes with the operation performed on the affected abstract states: Instead of removing a subgraph of abstract states from an ARG, we create a flat copy of all other abstract states, i.e., those abstract states that are not part of the subgraph.

To reduce the run time of the copy operation and the memory footprint for the copied abstract states, the *flat copy* keeps all internal data of abstract states untouched (e.g., information about program location, call stack, data state, etc.) and just references them from the new abstract states. This perfectly matches the *copy-on-write* idea and also the internal data structure of our framework, where abstract states consist of separate components for separate domains. Only those components where data needs to be changed are effectively constructed again, the rest is just referenced. This approach has two benefits:

- There is no need to implement and execute methods for copying internal data of abstract states (predicates, variable assignments, program counter, call-stack information, ...). Thus our new approach can easily be applied to all existing analyses.
- The *copy-on-write* approach has only a small memory overhead, because only new ARG states are constructed, the internal data of abstract states are shared and do not require additional memory.

6 Evaluation

Next we give evidence that the improvements in our refinement approach (no holes in the ARG) do not lead to significant performance drawbacks.

Benchmark Set. We evaluate our new *copy-on-write* refinement approach on a large subset of the SV-benchmark suite [3] containing over 5 500 verification tasks and compare it with the existing *in-place* approach.

Setup. We run all our experiments on computers with Intel Xeon E3-1230 v5 CPUs with 3.40 GHz, and limit the CPU time to 15 min and the memory to 15 GB. We use our implementation in CPAchecker [4] in revision `r29066`. The time needed for parsing the input program and exporting data is rather small compared to the analysis time, thus we measure the complete CPU time for the verification run of CPAchecker (i.e., including parsing, analysis, and witness export).

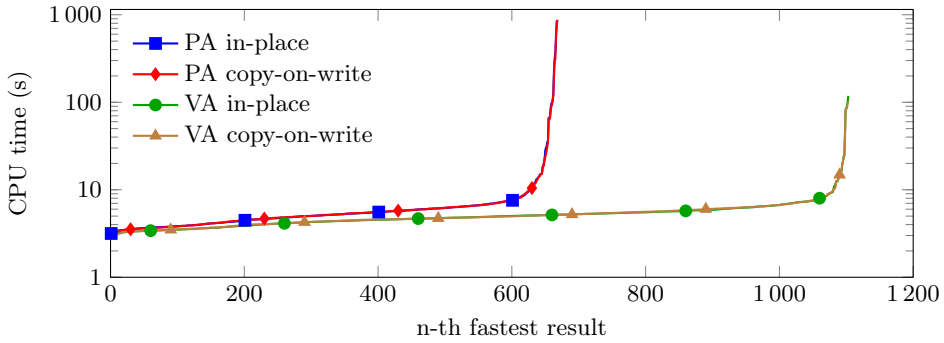
[3] `https://github.com/sosy-lab/sv-benchmarks`
[4] `https://cpachecker.sosy-lab.org`

Analysis Configuration. BAM can be combined with several analyses and for our experiments, we choose two combinations that are used in practice: BAM with predicate analysis (PA) and BAM with value analysis (VA) [2]. We configure BAM to use function and loop bodies as blocks, and predicate analysis computes abstractions, just as in the original work [27]. The expressive power of the program analysis depends only on the expressiveness of the predicate analysis or value analysis, and is not influenced by BAM. Except for the refinement approach itself, we do not change any configuration for each of the analyses. Thus, each of analyses should give the same verification answer in both cases.

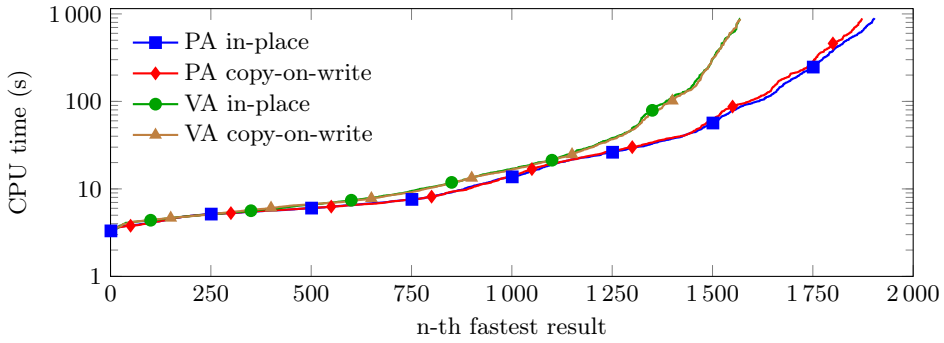
Results and Discussion. The experiments show nearly no difference in CPU time and also no significant difference in memory consumption between the two refinement approaches for each analysis. The reason for this result in terms of run time is that *copy-on-write* is extremely efficient and the light overhead is compensated by savings for recomputing missing parts of the state space. The reason for the same memory footprint is that the memory overhead for the additional ARGs is very small compared to the shared data (e.g., formulas for tracking variables). Note that memory usage is not fully predictable in general, as the Java garbage collection is applied non-deterministically.

The quantile plots in Fig. 4 show how many tasks are solved correctly with each of the approaches and each of the analyses. Figure 4a presents the results for all correctly solved verification tasks with low number of refinements (≤ 1): no difference in the results is visible for the two approaches per underlying analysis. For a low number of refinements, the equality of the results for different refinement approaches was expected, because the effect of missing block abstractions depends on a sufficiently large number of refinements. With only zero or one refinement the new approach behaves exactly as the *in-place* approach. With a growing number of refinements, the analysis could in principle perform differently. Figure 4b shows the CPU time for all correctly solved verification tasks where more than one refinement was needed. Both refinement approaches perform very similar, e.g., keeping block abstractions using *copy-on-write* is as good as recomputing missing block abstractions for both underlying analyses. (The similar performance of predicate analysis and value analysis is a coincidence, because both analyses use completely different techniques to track variables, assignments, and relations.)

Table 1 shows statistics about all verification results, for both approaches. There are some cases (for both predicate analysis and value analysis), where the analysis with one refinement approach delivers a result while the other does not. Sometimes eager application of a refined precision is beneficial, sometimes the overhead for recomputation of a missing block abstraction is too expensive. While the difference for value analysis is negligible, predicate analysis performs better with the *in-place* refinement, but needs more refinements than with *copy-on-write*. It seems that predicate analysis reacts much more fragile to changes in the refinement strategy and application of refined precisions than value analysis.



(a) CPU time of refinement approaches of BAM, ≤ 1 refinements only (plots are identical, because the changed approach does not affect the analysis)

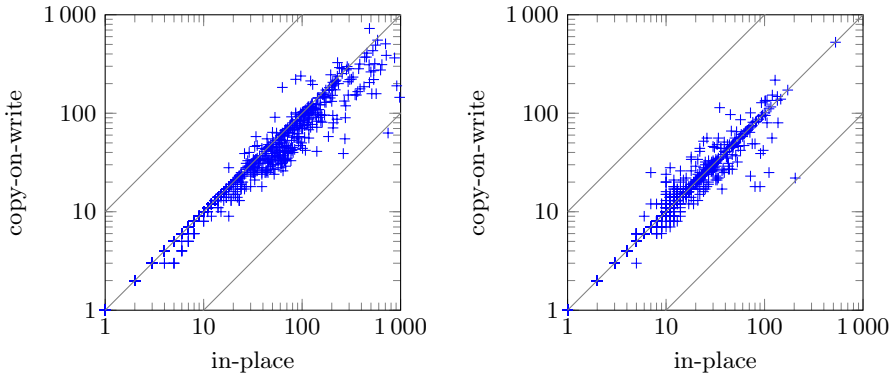


(b) CPU time of refinement approaches of BAM, ≥ 2 refinements only (plots only differ for predicate analysis with a larger run time, e.g., over 200 s, because the changed approach only affects the analysis if several blocks are accessed)

Fig. 4: Quantile plots for CPU time of refinement approaches

	Predicate Analysis		Value Analysis	
	In-place	Copy-on-write	In-place	Copy-on-write
Found proofs	2 149	2 121	2 352	2 352
Found bugs	425	422	322	322
Incorrectly found proofs	2	2	0	0
Incorrectly found bugs	0	0	2	2
Solved by only one approach	40	9	4	4
Avg. no. of refinements	53.7	19.4	8.21	8.35

Table 1: Statistics of refinement approaches of BAM



(a) #Refinements for predicate analysis **(b)** #Refinements for value analysis

Fig. 5: Comparison of *in-place* and *copy-on-write* refinement approach for predicate analysis and value analysis

Figures 5a and 5b compare the number of needed refinements for each solved task using scatter plots. The number of refinements includes also cases where a missing block abstraction has to be recomputed (recomputation is lazy and only applied if an error path with a *hole* was found; thus it counts as refinement, too). For predicate analysis with the *copy-on-write* approach, the majority of results is computed with a smaller number of refinements than with the *in-place* refinement: on average the new approach needs only a third of the refinements. For value analysis there is no clear difference in the number of refinements and the average number of refinements is also similar.

Threats to Validity. Our evaluation uses a large publicly available benchmark suite of C verification tasks in order to optimize the diversity in size and type of programs. While it seems clear that the concepts and results can be transferred to other verification tasks, such a claim is not backed up by our experiments. Besides the internal structure of a verification task, there are other factors that influence the behavior of an analysis. Thus, the external validity of the experiments regarding the application of refinements and precision updates is increased by the large number of experiments on different tasks. The chosen time limit of 15 min and memory limit of 15 GB for verifying a given task is inspired by the research community on software verification (cf. one of the reports on the International Competition on Software Verification [5]). Of course, the evaluation of our approach depends on the tool where it is implemented. To our knowledge, there is no other tool directly implementing the approach of BAM.

7 Conclusion

We developed a new approach for CEGAR-based refinement of block summaries that is based on *copy-on-write*. The new approach makes it possible to con-

struct an abstract reachability graph without holes, such that at the end of the program analysis, a complete proof is available to the user. The proof can be dumped for inspection, or a correctness witness can be extracted from the proof. We designed and implemented the *copy-on-write* refinement and provide a ready-to-use implementation in the framework CPAchecker. Re-using existing underlying analyses is possible without any further development overhead. The experimental comparison showed that there is almost no performance overhead for copy-on-write. Furthermore, the experimental comparison of the existing *in-place* with the new *copy-on-write* refinement strategy revealed interesting insights into some aspects of block summarization. In the future, we plan to design a parallel version of BAM to utilize a network of computers for our domain-independent analysis technique (cf. Swarm [21]): The new immutable block abstractions might also be beneficial in the context of resource-intensive communication between nodes of a computer network.

References

1. P. Andrianov, K. Friedberger, M. U. Mandrykin, V. S. Mutilin, and A. Volkov. CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 355–359. Springer, 2017.
2. P. Andrianov, V. S. Mutilin, M. U. Mandrykin, and A. Vasilyev. CPA-BAM-Slicing: Block-abstraction memoization and slicing with region-based dependency analysis - (competition contribution). In *Proc. TACAS*, LNCS 10806, pages 427–431. Springer, 2018.
3. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. SPIN*, LNCS 1885, pages 113–130. Springer, 2000.
4. T. Ball and S. K. Rajamani. The Slam project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
5. D. Beyer. Software verification with validation of results (Report on SV-COMP 2017). In *Proc. TACAS*, LNCS 10206, pages 331–349. Springer, 2017.
6. D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Correctness witnesses: Exchanging verification results between verifiers. In *Proc. FSE*, pages 326–337. ACM, 2016.
7. D. Beyer and K. Friedberger. Domain-independent multi-threaded software model checking. In *Proc. ASE*, pages 634–644. ACM, 2018.
8. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.
9. D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*, pages 29–38. IEEE, 2008.
10. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
11. D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.
12. D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.
13. D. Beyer, S. Löwe, and P. Wendler. Refinement selection. In *Proc. SPIN*, LNCS 9232, pages 20–38. Springer, 2015.

14. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Softw. Eng.*, 30(6):388–402, 2004.
15. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
16. E. M. Clarke, D. Kröning, N. Sharygina, and K. Yorav. SatAbs: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS*, LNCS 3440, pages 570–574. Springer, 2005.
17. K. Friedberger. CPA-BAM: Block-abstraction memoization with value analysis and predicate analysis (competition contribution). In *Proc. TACAS*, LNCS 9636, pages 912–915. Springer, 2016.
18. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Proc. PLDI*, pages 1–13. ACM, 2004.
19. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
20. C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, pages 102–116. Springer, 1971.
21. G. J. Holzmann, R. Joshi, and A. Groce. Tackling large verification problems with the Swarm tool. In *Proc. SPIN*, LNCS 5156, pages 134–143. Springer, 2008.
22. R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.
23. K. L. McMillan. Lazy abstraction with interpolants. In *Proc. CAV*, LNCS 4144, pages 123–136. Springer, 2006.
24. T. W. Reps. Program analysis via graph reachability. In *Proceedngs of the 1997 International Symposium on Logic Programming (ILPS'97)*, pages 5–19. MIT, 1997.
25. O. Sery, G. Fedyukovich, and N. Sharygina. Funfrog: Bounded model checking with interpolation-based function summarization. In *Proc. ATVA*, LNCS 7561, pages 203–207. Springer, 2012.
26. O. Sery, G. Fedyukovich, and N. Sharygina. Interpolation-based function summaries in bounded model checking. In *Proc. HVC*, LNCS 7261, pages 160–175. Springer, 2012.
27. D. Wonisch and H. Wehrheim. Predicate analysis with block-abstraction memoization. In *Proc. ICFEM*, LNCS 7635, pages 332–347. Springer, 2012.
28. Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *TOPLAS*, 29(3):16, 2007.