Adjustable Block Analysis: Actor-based Creation of Block Summaries for Scaling Formal Verification

Master's Thesis in Computer Science

22.02.2022

Ludwig-Maximilians-Universität München

Matthias Kettl

Supervisor: Prof. Dr. Dirk Beyer
Mentor: Thomas Lemberger

Abstract

Software quality assurance becomes more and more popular in modern software development. Fixing bugs early, reduces the cost and increases the quality of the product. Since a great portion of time is needed for debugging, assisting techniques are integrated. Many techniques use formal verification as a basis. Although formal verification is applied with great success to many software development pipelines, the scalability is still a big challenge. In this work, we introduce the concept of distributed CPAs (configurable program analyses). The core idea is to partition the control flow automaton (CFA) of a given input program in coherent blocks with one entry and one exit node. Workers analyze every block parallely and broadcast new information to every other worker. If a block contains an error location, the block is analyzed backwards from the error location. Whenever the analysis reaches the top of the block, we ask whether other blocks can prove the error location (un)reachable. For this, we reuse the results of previous verification runs on other blocks. Our approach is easily extensible to support any existing configurable program analysis and additionally allows an easy integration of other concepts, e.g., fault localization. This work shows and explains the implementation of a framework for distributed CPAs in CPAchecker. Furthermore, we perform a thorough evaluation of the approach, showing that our implementation is sound and matches the results of the existing predicate analysis. Distributing the work to many workers improves the speed of the analysis compared to running the complete analysis on only one worker. However, the great amount of transferred data between workers and additional SAT-checks slow down the verification process causing more out-of-memory errors and timeouts compared to the predicate analysis. Nonetheless, the evaluation reveals the current bottle-necks and points us to potentially useful and effective improvements for the future.

Contents

List of Algorithms

List of Figures

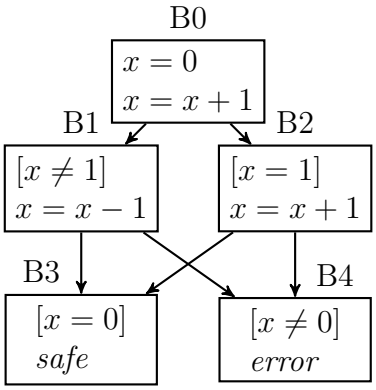
List of Tables

1 Introduction

Reducing the number of bugs in a program is one of the biggest challenges in modern software development as the programs grow in size and the time restrictions are dense [14]. In recent years, many techniques to tackle the problem developed. Verification tools prove programs on certain properties. In case of a property violation, developers are alarmed and try to fix the problem. On top of that, fault localization and automatic program repair techniques are integrated in a rising number of projects to improve the software quality. Although the computation power increased, many programs are to complex to be examined in a reasonable amount of time, not to mention the additional time needed for fault localization and other techniques that rely on verification results. In this work, we introduce the concept of a distributed configurable program analysis to reduce the wall time of program verification. The goal is to formalize and implement a concept to verify programs distributed on many workers. In our approach, every worker verifies a coherent subspace of a program, called *code block* and communicates new results to all other workers in the manner of an actor model [15]. We consider every worker as an entity of the actor model, having a pre- and a post-condition. With the help of the pre- and post-conditions, workers summarize their knowledge of their *code block* to subsequently generate new information, needed by other workers. The pre- and the post-condition summarize the knowledge of directly dependent *code blocks*. Every worker analyzes its *code block* in two directions. Updates to the precondition cause a forward analysis of the current block. Successive *code blocks* use the result of the forward analysis again to update their precondition. If a forward analysis reaches an error location, it starts a backward analysis from that location in the input program. The worker communicates the result of the backward analysis and asks them whether the error location is actually reachable. In case, no block responses with another backward analysis, the program is safe. Otherwise, we trigger the backward analysis on this *code block* and repeat the procedure. However, if a satisfiable error condition reaches the program entry point, we have proven the program unsafe. Figure 1 shows the decomposi-

```
1  int x = 0;
2  x++;
3  if (x == 1) {
4      x++;
5  } else {
6      x--;
7  }
8  assert(x==0);
```

(a) Program



(b) Code blocks

Figure 1: *Code blocks* for a given program

tion of the given program into 5 blocks. Every program statement maps to exactly one *code block*. After the decomposition, we spawn a worker for every block. The workers start to analyze the blocks parallely. Initially, they start with a forward analysis and the precondition *true*. Whenever the worker for B0 finishes, it broadcasts the message $x^{\langle 0 \rangle} = 0 \wedge x^{\langle 1 \rangle} = x^{\langle 0 \rangle} + 1$. Subsequently, the workers for blocks B1 and B2 update their precondition and schedule a new forward analysis resulting in a stronger precondition for their respective successors. The worker analyzing B4 runs into an error. Hence, it starts a backward analysis and broadcasts the *error condition* $x^{\langle 0 \rangle} \neq 0$. Now, the workers for B1 and B2 update their post-condition to $x^{\langle 0 \rangle} \neq 0$ and schedule a backward analysis themselves.

We implement the described concept in CPAchecker a framework for formal verification. Moreover, CPAchecker [7] is an excellent tool [2] for formal verification. The components of a configurable program analysis (CPA) fit our approach perfectly and allow an easy extension of any CPA to a distributed CPA (DCPA). The integration of BenchExec [21] allows an in-depth evaluation of the efficiency of our approach.

The following work explains the background knowledge needed for this work and defines the concept of a distributed CPA. Moreover, we share implementation details, improvements and possible other adaptions to our configurable approach. At the end, we compare the distributed approach to the default approach and point out the advantages and disadvantages. Additionally, we showcase the expandability of our technique by distributing an already implemented fault localization algorithm.

2 Related Work

2.1 BAM

Block-abstraction memoization (BAM) [22] decomposes the control flow automaton (Definition 1) in arbitrary blocks. The decomposition is given by the user. There is only one restriction to the blocks: they have to be nested or pairwise disjoint. In other words, blocks B' and B'' only share common locations if all locations of B' are contained in B'' or vice versa. Usually, blocks summarize, e.g., function calls. The analysis can now calculate the abstract reachability tree (ART) of each block and reuse known results later. For this, every block is considered as one entity. The transfer relation (Definition 9) supports blocks as single entities next to the standard transfer over edges of the CFA. The advantage of nested blocks lies in the abstraction. Inner blocks may depend on less variables and thus the precision of abstract states of parent blocks can be reduced to a smaller set of predicates to track. However, exiting the block requires a strengthening to restore the lost information. The approach has proven to increase the performance of the predicate analysis by caching results of nested blocks.

In our approach blocks only have one exit and one entry node. Our concept can be extended to blocks with multiple in- and output nodes but subsequently the workers need to deal with more complex cases. Pre- and post-conditions have to be stored separately for all in- and output nodes. Reusing results can still be done but we would have to query the correct messages instead of just taking the latest messages. In the future, we could expand our block structure to arbitrary coherent blocks but the complexity of the workers increases.

2.2 Infer

Another tool aiming at scalable verification is Infer[1] from Facebook. Today, Infer is also developed by the community. Infer is an incremental static analyzer integrated in the continuous integration of projects. Static analyzers try to prove programs (un)safe without executing them. The tool is widely used across projects of prestigious companies. The state space of a program grows with the complexity of the code. Still, developers need the verification result within approximately 10 minutes to continue the natural workflow[2]. To rapidly report verification results, Infer implements separation logic [19]. Separation logic allows to reason on small parts of a program instead of having the complete source code under analysis. Additionally, Infer stores results of previous runs and reuses them whenever it is possible. The incremental approach scales well with big projects. Infer will not report bugs twice. In case the developers ignore a reported bug, the tool will not display it again, minimizing repeated reports of false alarms, again contributing to the scalable approach.

2.3 SynergiSE

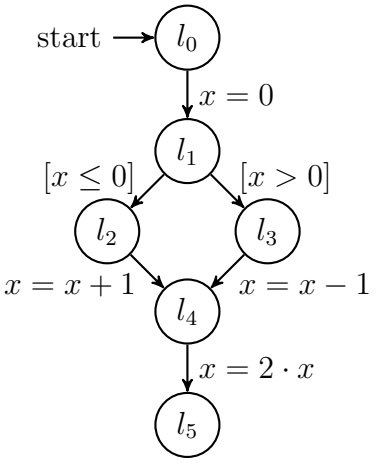
SynergiSE [20] distributes symbolic execution [18] to workers. Symbolic execution introduces symbolic values for variables and tracks path constraints over these symbolic values. The path constraints can be passed to an SMT-solver that performs a SAT-check. Depending on the result of the SAT-check, the path is deemed (in)feasible. SynergiSE explores symbolic paths to a given depth. In case, a path is feasible, it assigns explicit values to every symbolic variable resulting in a set of test-inputs. Running the program with the calculated values as initial variable assignment is guaranteed to reach the explored path to the given depth. Thus, we obtain a one-to-one mapping of test-inputs to paths. Deepening the feasible ranges can now be distributed over different workers. Whenever a worker succeeds, the results can be shared across the workers. If a worker receives a test-input t and a given depth d then it knows by definition that the path, reached with the initial assignment t is feasible up to depth d. The worker does not need to perform any SAT-checks allowing efficient sharing of solved path constraints.

[1] `https://fbinfer.com/`

[2] `https://engineering.fb.com/2015/06/11/developer-tools/open-sourcing-facebook-infer-identify-bugs-before-you-ship/`

```
1  int x = 0;
2  if (x <= 0) {
3      x++;
4  } else {
5      x--;
6  }
7  x = 2 * x;
```

(a) Program



(b) CFA

Figure 2: CFA for a given program

3 Background

3.1 Control Flow Automaton (CFA)

The *control flow automaton* (CFA) is a directed, potentially circular graph, representing a program.

Definition 1 (Control Flow Automaton) Formally, the CFA is a triple (L, l_0, G) where L is the set of program locations, $l_0 \in L$ denotes the start location and G contains all possible transition between the locations in L. An edge $g = (l, o, l') \in G$ is an element of $L \times O \times L$, meaning that we can reach l' through l by executing o where $o \in O$ resembles an operation, i.e., statements like `x = x + 1;` or assumes like `[x < 2]`.

The following work uses *CFA node* synonym to a location $l \in L$ and *CFA edge* synonym to a transition $g \in G$. Figure 2 shows a CFA on the right-hand side for the given program on the left-hand side. Every edge represents

one transition, e.g., the assignment $x = 0$ (first edge). The CFA branches, whenever we reach conditional statements (if, for, while). The if-statement in line 2 causes a branching because there are two possible paths, either for $x \leq 0$ or $x > 0$. For the construction of CFAs it does not matter whether the path is actually feasible. In case our program contains functions called multiple times, we inline the CFA of the function at the corresponding positions to treat the program like no function calls exists.

3.2 Decomposition of CFAs

Code blocks are the unit we deal with in the further work. Every code block is a coherent subgraph of the CFA. Hence, they are directed and perhaps circular, too.

Definition 2 (Code Block) A *code block* $B = (L_B, l_{B_0}, l_{B_f}, G_B)$ is a subgraph of the CFA (L, l_0, G) where $L_B \subseteq L$ is the set of locations, $l_{B_0} \in L_B$ symbolizes the initial location and $l_{B_f} \in L_B$ symbolizes the final location of that specific block. $G_B = \{(l, *, l') | (l, *, l') \in G \wedge l, l' \in L_B\}$ contains all transitions of the block. Having exactly one start and one final location ensures that every block has only one entry and one exit node by definition. A *code block* is coherent, meaning all locations in L_B are connected over transitions in G_B.

Definition 3 (Predecessor of a Code Block) Block $B = (*, *, l_{B_f}, *)$ is a predecessor of $B' = (*, l_{B'_0}, *, *)$ if $l_{B_f} = l_{B'_0}$. The function $pred(B)$ returns all predecessors of a block.

Definition 4 (Successor of a Code Block) Block $B = (*, l_{B_0}, *, *)$ is a successor of $B' = (*, *, l_{B'_f}, *)$ if $l_{B_0} = l_{B'_f}$. The function $succ(B)$ returns all successors of a block.

We can decompose the CFA in an arbitrary number of *code blocks* $\mathcal{B}$. There are two kinds of decompositions: strict and complete.

Definition 5 (Complete Decomposition of the CFA) We consider the decomposition to be complete, if the union of the edges of all blocks $B = (L_B, l_{B_0}, l_{B_e}, G_B) \in \mathcal{B}$ contains all edges of the original CFA (L, l_0, G). Formally, if $\bigcup_{B \in \mathcal{B}} G_B = G$ holds.

Definition 6 (Strict Decomposition of the CFA) We consider the decomposition to be strict, if the intersection of the edges of all blocks $B = (L_B, l_{B_0}, l_{B_e}, G_B) \in \mathcal{B}$ is empty. Formally, if

$$\forall B, B' \in \mathcal{B} : B \neq B' \Rightarrow G_B \cap G_{B'} = \emptyset$$

Table 1: Possible values for each part of a message

Message part	possible values
τ	BlockPostcondition, ErrorCondition, ErrorConditionUnreachable, Result, Error
id	unique string identifying a block
l_{id}	CFA node ID as integer
ρ	any JSON string

holds.

The following work assumes that every decomposition is both, complete and strict, meaning that we can map every edge of the CFA to exactly one block. Locations, on the other hand, will be part of multiple blocks.

Definition 7 (Real Decomposition of the CFA) We consider the decomposition to be real if it is strict and complete.

3.3 Messages

In our actor model, the entities communicate over messages. Each message consists of four parts: the message type τ, the ID of the sender id, the ID of a target node l_{id} and the payload ρ.

Definition 8 (Message) Formally, a message M can be written as a four-tuple (τ, id, l_{id}, ρ)

Table 1 lists all possible values for each part of the message. The id is a unique string mapping to a *code block*. The message type τ has one of five values indicating how the underlying analysis should process the message. *Code blocks* react differently to every message type. Messages indicate the location they originate from by providing the location ID stored in l_{id}. We assume that every location can be identified by an integer. The payload ρ is an arbitrary JSON string varying in the structure based on the type τ. Usually, ρ contains a key-value pair for each distributed CPA which brings us to the following two sections, defining (distributed) CPAs.

3.4 Configurable Program Analysis (CPA)

Definition 9 (CPA) A configurable program analysis (CPA) [6] consists of an abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator merge and a

stop operator stop. Formally, we write a CPA $\mathbb{C}$ as a four-tuple $(D, \rightsquigarrow, \text{merge}, \text{stop})$. A CPA operates on a CFA (L, l_0, G). The following paragraphs explain each of the four components in more detail.

3.4.1 Lattice

Before we continue with the definition of the components, we have to define lattices. Lattices are based on a partial order $\sqsubseteq$ and operate on a set E. A partial order is reflexive ($\forall e \in E : e \sqsubseteq e$), transitive ($e \sqsubseteq e' \wedge e' \sqsubseteq e'' \Rightarrow e \sqsubseteq e''$) and antisymmetric ($e \sqsubseteq e' \wedge e' \sqsubseteq e \Rightarrow e = e'$). If we can find a least upper bound in E for every possible subset of E, then $(E, \sqsubseteq, \sqcup, \top)$ is a semi-lattice. For two elements $e, e' \in E$ the operator $\sqcup$ yields the least upper bound of both elements. The top element $\top$ is an upper bound for every pair of elements in E. Formally, we can define it as $\top = \bigsqcup E$.

3.4.2 Abstract Domain

The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ is defined over a set of concrete states C, a semi-lattice $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$ and a concretization function $[\![\cdot]\!] : E \mapsto 2^C$ that maps an abstract state to a set of concrete states. The set E of our semi-lattice represents the abstract states of the program under analysis. Concrete states are states that our program can actually reach. Thus, an abstract state may represent multiple concrete states since the abstraction over-approximates.

3.4.3 Transfer Relation

The transfer relation

$$\rightsquigarrow \subseteq E \times G \times E$$

computes all possible abstract successors of an abstract state when transitioning over an edge $g \in G$.

3.4.4 Merge Operator

The merge operator

$$\text{merge} : E \times E \mapsto E$$

combines two abstract states to one abstract state without loosing any information. In fact, the new abstract state is an over-approximation of both input states if the first abstract state is not subsumed by the second abstract state. The merge operator is based on $\sqcup$. If we do not want to merge at all, we can define $merge_{sep}(e_1, e_2) = e_2$.

Algorithm 1: CPA adapted from [6]

Input: $(D, \rightsquigarrow, \text{merge}, \text{stop})$: CPA
Input: e_0: with $e_0 \in E$ and E the lattice of D
Input: *cfa*: CFA, needed for $\rightsquigarrow$
Output: R: reached set
Result: all reachable abstract states

```
1  waitlist = {e_0};
2  reached = {e_0};
3  while waitlist ≠ {} do
4      choose e from waitlist;
5      waitlist = waitlist \{e};
6      for each e' with e ⇝ e' do
7          for each e'' ∈ reached do
8              e_new = merge(e', e'');
9              if e_new ≠ e'' then
10                 waitlist = (waitlist ∪{e_new} \ {e''});
11                 reached = (reached ∪{e_new} \ {e''});
12         if ¬stop(e', reached) then
13             waitlist = waitlist ∪{e'};
14             reached = reached ∪{e'};
15 return R;
```

3.4.5 Stop Operator

The stop operator

$$\text{stop} : E \times 2^E \mapsto \mathbb{B}$$

decides whether a newly computed abstract state should be put in the waitlist. The decisions is made with the help of a set of already reached states. One possible implementation of a stop operator does not put new abstract states in the waitlist if one of the contained states subsumes the current state.

3.4.6 CPA Algorithm

For completeness, we briefly present the CPA algorithm (from [6]). We start with the waitlist and the reached set containing the inital element e_0. As long as the waitlist is not empty, we remove an element of it and compute the abstract successors of it with the help of our transfer relation. Afterwards, we merge every successor e' with every existing element in the reached set.

We replace the existing element if the merge operator returns a new element. The stop operator decides whether the current successor should be enqueued in the waitlist.

3.4.7 Location CPA

We briefly cover the most important aspects of the location CPA ($\mathbb{L}$). $\mathbb{L}$ transfers from one location l_i to another location l_j if there exists an edge $(l_i, *, l_j) \in G$ in the CFA. With its help, we can track the current location of abstract states. From now on, we assume that we know the exact location of every abstract state produced by, e.g., the predicate analysis.

3.4.8 Predicate Analysis

In this work we focus on the predicate analysis with adjustable-block encoding (ABE) although the presented work can easily be adapted to every kind of CPA by extending it to a distributed CPA described in Section 3.7. We will now elaborate the definition of the predicate analysis as an own CPA by defining all the components of a CPA. We take the definition from [8].

Abstract Domain

The predicate abstract state $(l, \psi, l^\psi, \phi) \in (\mathbb{L} \cup \{l_\top\}) \times \mathcal{P} \times (\mathbb{L} \cup \{l_\top\}) \times \mathcal{P}$ is a four-tuple, where l represents the current location, l^ψ the location of the last abstraction, ψ a Boolean combination of predicates in our precision π and ϕ a (disjunctive) path formula. $\mathcal{P}$ denotes the set of predicates. The location $l_\top$ is the top element of a semi-lattice over the program locations. Hence, the least upper bound for any two distinct locations equals $l_\top$. The reflexive property implies that the least upper bound location of equal locations is again that location. The top element of predicate abstract states equals $(l_\top, \mathit{true}, l_\top, \mathit{true})$. We order the abstract states $e' = (l_1, \psi_1, l_1^\psi, \phi_1)$ and $e'' = (l_2, \psi_2, l_2^\psi, \phi_2)$ as follows:

$$e' \sqsubseteq e'' \Leftrightarrow (e'' = \top) \vee ((l_1 = l_2) \wedge (\psi_1 \wedge \phi_1 \Rightarrow \psi_2 \wedge \phi_2)).$$

The concretization function maps an abstract state to concrete states that fulfill ϕ. The precision π is a finite set of predicates that is implied by actual executions of the program at the current location.

Transfer Relation

The transfer relation contains all triples (e, g, e') for $g = (l_1, o, l_2) \in G$, $e = (l_1, \psi_1, l_1^\psi, \phi_1)$ and $e' = (l_2, \psi_2, l_2^\psi, \phi_2)$ such that the the following holds:

$$\begin{cases} \phi_2 = \mathit{true} \wedge (\psi_2 = (\mathrm{SP}_o(\phi_1 \wedge \psi_1))^{\pi(l_2)}) \wedge l^{\psi_2} = l_2 & \text{if } \mathrm{blk}(e, g) \vee (l' = l_E) \\ (\psi_2 = \mathrm{SP}_o(\phi)) \wedge (\psi_2 = \psi_1) \wedge (l_2^\psi = l_1^\psi) & \text{otherwise} \end{cases}.$$

The strongest post operator SP returns the strongest Boolean combination of predicates in π at a specific location when transferring with the operation o. The user decides when the block operator blk returns *true*. Either, blk returns true after a given number of operations or on certain locations, e.g., locations right before function calls or loop heads.

Merge Operator

The merge operator for $e' = (l_1, \psi_1, l_1^\psi, \phi_1)$ and $e'' = (l_2, \psi_2, l_2^\psi, \phi_2)$ is defined as follows:

$$\mathrm{merge}(e', e'') = \begin{cases} (l_2, \psi_2, l_2^\psi, \phi_1 \vee \phi_2) & \text{if } (l_1 = l_2) \wedge (\psi_1 = \psi_2) \wedge (l_1^\psi = l_2^\psi) \\ e'' & \text{otherwise} \end{cases}.$$

Two states are combined if they are on the same location and equal abstractions were computed on the same location.

Stop Operator

The stop operator checks if any state in the reached set already covers the current state. In case, one of the abstract states in the reached set does, the stop operator does not add the current state to the waitlist.

3.4.9 Analysis Direction

Since the CPA runs on the CFA (L, l_0, G), we can simply flip all edges and construct a new CFA^T (L, L_F, G') with $G' = \{(l', *, l) | (l, *, l') \in G\}$. L_F marks the former last location as the new initial location. CFA^T allows the execution of a backwards analysis. In reality, there are more adaptions to make to enable a backwards analysis. For the sake of simplicity, we omit these adaptions and assume that any CPA can run backwards if we flip the edges of a CFA. Analogously, *code blocks* can be flipped, too. We denote the reversal of a block B with B^T.

Table 2: Calculation of SSA-indices for forward analysis

SSA-map	code	path formula	updated SSA-map
$\{x:0\}$	x = x + c;	$x^{\langle 1\rangle} = x^{\langle 0\rangle} + c$	$\{x:1\}$
$\{x:0, y:0\}$	x = x + y;	$x^{\langle 1\rangle} = x^{\langle 0\rangle} + y^{\langle 0\rangle}$	$\{x:1, y:0\}$
$\{x:0, \dots\}$	x = v;	$x^{\langle 1\rangle} = v$	$\{x:1, \dots\}$
$\{x:0, \dots\}$	[x == x + v]	$x^{\langle 0\rangle} = x^{\langle 0\rangle} + v$	$\{x:0, \dots\}$

Definition 10 (Reversed Block) Formally, $B^T = (L_B, l_{B_f}, l_{B_0}, G'_B)$ for a block $B = (L_B, l_{B_0}, l_{B_f}, G_B)$, where $G'_B = \{(l', *, l) | (l, *, l') \in G_B\}$.

Nevertheless, one distinction has to be made. The path formulas have to be built differently, when traversing the block backwards. The next section explains what path formulas are and how they are adapted.

3.5 Static Single Assignment

3.5.1 Forward Analysis

The predicate analysis computes path formulas for all possible paths in a CFA. These formulas can be constructed rooted on the operation o of an edge $(l, o, l') \in G$. To keep it simple, we limit the set of operations O to assumes (e.g., [x == 5]) and arithmetic statements (e.g., x = x + y;). The static single assignment map (SSA-map) maps the current static single assignment index (SSA-index) to the corresponding variable. Initially, all variables are indexed with 0. After a new value is assigned to a variable, the index increases by 1. We denote a constant value with c and a constant value or a variable with the letter v. We need the SSA-indices to let SMT-solvers prove formulas (un)satisfiable. Without SSA-indices, the path formula $x = x + 1$ already becomes unsatisfiable because there exists no number that equals its successor. With SSA-indices the path formula equals $x^{\langle 1\rangle} = x^{\langle 0\rangle} + 1$ and we can provide a satisfying assignment, e.g., $x^{\langle 1\rangle} = 1 \wedge x^{\langle 0\rangle} = 0$. Table 2 gives four examples. Assume statements (fourth row in Table 2) do not change SSA-indices. Whenever we assign new values to a variable, we increase the index by 1. The right-hand side remains unchanged (c.f., third row in Table 2).

Table 3: Calculation of SSA-indices for backward analysis

SSA-map	code	path formula	updated SSA-map
$\{x:0\}$	x = x + c;	$x^{\langle 0\rangle} = x^{\langle 1\rangle} + c$	$\{x:1\}$
$\{x:0, y:0\}$	x = x + y;	$x^{\langle 0\rangle} = x^{\langle 1\rangle} + y^{\langle 0\rangle}$	$\{x:1, y:0\}$
$\{x:0, \dots\}$	x = v;	$x^{\langle 0\rangle} = v$	$\{x:1, \dots\}$
$\{x:0, \dots\}$	[x == x + v]	$x^{\langle 0\rangle} = x^{\langle 0\rangle} + v$	$\{x:0, \dots\}$

3.5.2 Backward Analysis

For the backward analysis, the construction of the SSA-indices is slightly different. While assume edges are handled in exactly the same way as shown in Table 2, arithmetic statements now assign the increased index on the right-hand side instead of the left-hand side. Contrary to the forward analysis, assignments where the variable does not occur on the right-hand side of the assignment stay unchanged regarding the SSA-indices. However, we increase the SSA-index of the variable afterwards (c.f., third row in Table 3, the index is 1 but the formula is instantiated with 0).

3.6 SMT Solvers and Models

SMT solvers (like MathSat5 [11] or Z3 [13]) check formulas for satisfiability. A formula is satisfiable if there exists a value for every variable in the formula without provoking a contradiction. There may be infinite possibilities. Contrary, if the SMT solver does not find a solution, the formula is considered unsatisfiable. A model of a formula is one concrete example of a valid variable assignment. The formula *false* is unsatisfiable by definition. Hence, *true* is satisfiable by definition. We continue with an example for an non-trivial unsatisfiable formula:

$$x^{\langle 0\rangle} < 5 \wedge x^{\langle 1\rangle} = x^{\langle 0\rangle} + 1 \wedge x^{\langle 1\rangle} = 6.$$

There are no values for $x^{\langle 0\rangle}$ and $x^{\langle 1\rangle}$ that satisfy the formula since the addition of 1 to a number less than 5 never equals 6. One small change makes the formula satisfiable:

$$x^{\langle 0\rangle} \leq 5 \wedge x^{\langle 1\rangle} = x^{\langle 0\rangle} + 1 \wedge x^{\langle 1\rangle} = 6.$$

The variable assignment $x^{\langle 0\rangle} = 5$, $x^{\langle 1\rangle} = 6$ does not provoke a contradiction and thus is a model of the formula. In fact, it is the only model for the given formula.

3.7 Distributed CPA (DCPA)

Definition 11 (DCPA) A DCPA (distributed configurable program analysis) is a five-tuple ($\mathbb{C}$, *serialize*, *deserialize*, *proceed*, *combine*) operating on one *code block* in $\mathcal{B}$. $\mathbb{C}$ denotes an arbitrary CPA with the semi-lattice $\mathcal{E}_{\mathbb{C}} = (E, \sqsubseteq, \sqcup, \top)$ as defined in Definition 9. In addition, we define $\mathcal{M}$ as the set of all possible messages (Definition 8). We extend $\mathbb{C}$ to a distributed CPA with the implementation of the four above-mentioned operators.

To avoid misunderstandings, we always use the term CPA to address the classic CPA defined in Definition 9 and the term distributed CPA or DCPA to address the distributed CPA as defined in Definition 11.

3.7.1 Serialize Operator

The *serialize operator*

$$\text{serialize} : E \mapsto \mathcal{M}$$

takes an abstract state and transforms it into a message m.

3.7.2 Deserialize Operator

The *deserialize operator*

$$\text{deserialize} : \mathcal{M} \mapsto E$$

takes a message and transforms it into an abstract state $e \in E$.

3.7.3 Combine Operator

The *combine operator*

$$\text{combine} : E \times E \mapsto E$$

takes two abstract states and computes one abstract state representing both states if possible. For some CPAs the *merge operator* of $\mathbb{C}$ suffices to combine states. The *combine operator* can be extended to accept sets of abstract states as input. The definition of combine_E is as follows ($\mathbb{E} \subseteq E$):

$$\text{combine}_E(\emptyset) = \top \tag{1}$$
$$\text{combine}_E(\{e_1\}) = e_1 \tag{2}$$
$$\text{combine}_E(\{e_1, e_2\}) = \text{combine}(e_1, e_2) \tag{3}$$
$$\text{combine}_E(\{e_1, e_2\} \cup \mathbb{E}) = \text{combine}_E(\{\text{combine}(e_1, e_2)\} \cup \mathbb{E}) \tag{4}$$

Figure 3: Definition of the *combine operator* over sets

Lines (1) and (2) show the base cases. The operator combine_E returns $\top$ if the given set is empty. For sets of size one, the operator just returns the one element. If a set consists of exactly two elements, we combine them using the default *combine operator* discussed above (3). For sets with at least 3 elements, we call combine_E recursively. The recursive call reduces the size of the input set by one as two of its states are combined with the default *combine operator*. The new state is re-added to the input set (4).

Changing the *combine operator* to return and accept sets of abstract states (combine: $2^E \mapsto 2^E$) instead of only combining two abstract states to one, is also possible. Our analysis will then consider multiple initial abstract states and compute successors for all of them.

3.7.4 Proceed Operator

The *proceed operator*

$$\text{proceed} : \mathcal{M} \mapsto \mathbb{B} \times 2^{\mathcal{M}}$$

takes a message and decides whether the distributed analysis should go on. It returns a Boolean value $v \in \mathbb{B} = \{\textit{true, false}\}$ together with a set of messages with which the block analysis should continue or stop. The *proceed operator* can be based on the stop operator of $\mathbb{C}$ if we deserialize the message first. The *proceed operator* has to return (*false*, *) if the given message represents an abstract state that cannot be reached via this block. This is, for example, the case if a message contains a predicate abstract state with the path formula *false*. Additionally, the *proceed operator* returns (*false*, *) whenever the target location l_{id} does not match the respective block entry point.

3.7.5 Loops

In some cases, DCPAs operate on blocks where the initial location equals the final location. This happens, for example, if the block contains a full loop

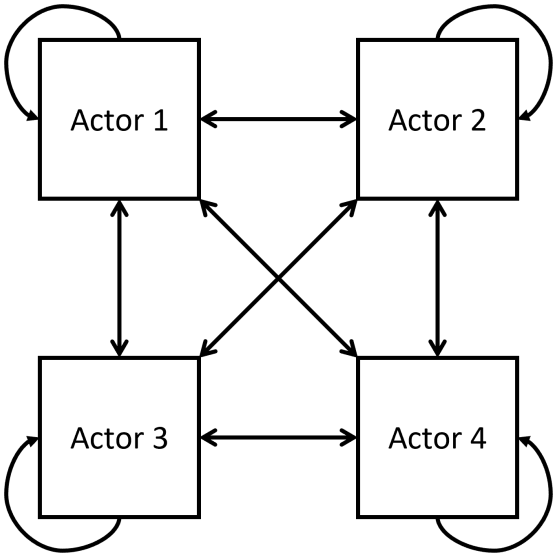


Figure 4: An actor model with 4 actors and 4^2 connections

of the CFA. DCPAs on loops trigger an endless forward analysis initially because they do not know the necessary information of the predecessors that satisfies the abort condition of the loop. To overcome this problem, DCPAs have the characteristic to stop whenever they reach the final location after the initial location has already been visited. Afterwards, they broadcast the new information. Since such blocks are their own predecessors and successors, they receive their own message. By deserializing their own message, they continue with one more iteration of the loop. This guarantees to unroll the loop as far as necessary. At some point, the precondition of the block is strong enough to exit the loop eventually, i.e., stop calculating new messages. Thus, neither the forward nor the backward analysis can ever enter an endless loop. In the worst case, a worker produces infinite messages but it will never be stuck in an endless loop.

3.8 Actor Model

In an actor model [15], every entity is an actor that broadcasts new information with a message to all other actors including itself. Thus, n actors have a total of n^2 connections. The messages are the unit of the actor model. If an actor receives a message it decides on its own whether the information is valuable. Respectively, the information is kept or discarded. Each actor works separately and independently on kept messages. Figure 4 shows an example with 4 actors and 16 connections.

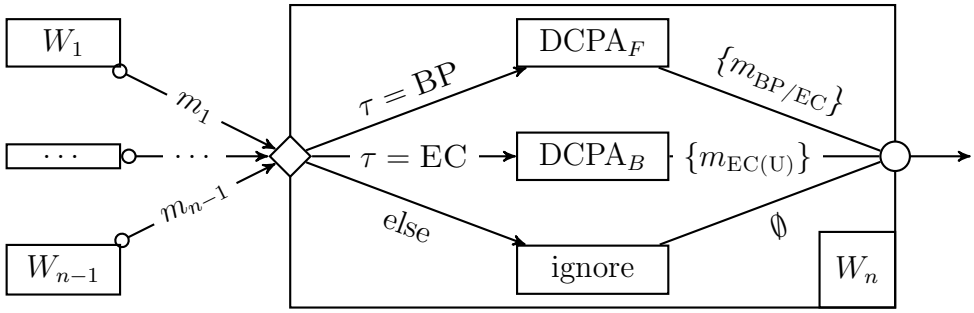


Figure 5: Abstract depiction of a worker. The worker receives $n-1$ messages from other actors and triggers different analyses based on the message type.

4 Actor-Based Block Summaries for Formal Verification

4.1 Distributed Framework

4.1.1 Basic Idea

The basic idea is that every worker verifies one block B, i.e., a subgraph of the CFA with a distributed CPA. All results of the analyses, i.e., abstract states are transmitted to all other workers that consequently adapt their pre- and post-conditions. First, we describe the structure of a worker with the help of Figure 5. On the left hand side, we see workers W_0 to W_{n-1} sending messages to worker W_n. W_n collects the messages at the entry point ($\diamond$). Afterwards, the worker processes every message one by one. Depending on the type τ of a message, the worker schedules different tasks. A message with type $\tau = \mathrm{BP}$ (for BlockPostcondition) is followed by a distributed forward analysis, resulting in either a new message of type BlockPostcondition or in a message of type ErrorCondition in case the forward analysis reaches an error location. For $\tau = \mathrm{EC}$ (for ErrorCondition) the worker runs a back-

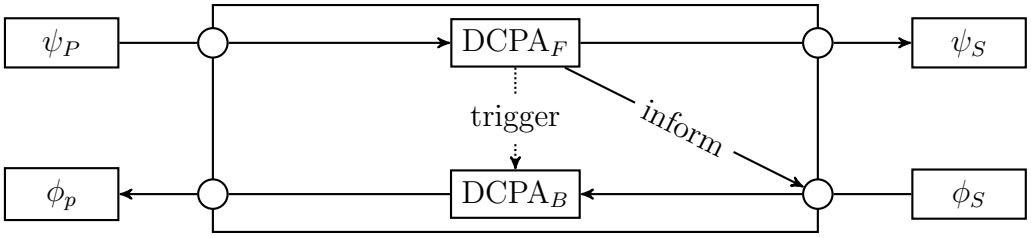


Figure 6: Simplified worker schema

ward analysis either resulting in a message telling the other blocks that the previous error condition is not reachable via this block or a new stronger error condition. Other messages are ignored or cause the worker to shutdown. To put it in another way, messages with the type $\tau = \text{BLOCKPOSTCONDITION}$ update the precondition ψ of a worker. Their combination via the *combine operator* implies what has to be valid at the initial location of a block. With that in mind, we can simplify the schema in Figure 5 to the schema depicted in Figure 6. Our worker has a precondition ψ_P that holds at the initial location of its block. The combination of the received messages of type $\tau = \text{BLOCKPOSTCONDITION}$ of all predecessor blocks form our precondition ψ_P. We only need the messages from our predecessor blocks since they contain the information for our initial location. Whenever our precondition is updated, we run a new forward analysis. Eventually, we get new abstract states at the final location of our block. The worker combines these states, serializes them and broadcasts them. The successors update their preconditions and run a forward analysis by themselves based on that message. Additionally, the forward analysis informs the backward analysis about its recent results. Later, we will see why. If the forward analysis reaches an error location, it triggers a backward analysis, starting from the error location and trying to reach the initial block location. The resulting message updates the post-conditions ϕ_P of all predecessors.

If a successor updates the post-condition of a worker, the worker runs a backward analysis. But first, it uses the already computed results from own forward analyses if they are already available (depicted by the arrow "inform" from DCPA$_F$ to the post-condition). Since the forward analysis already reached the final location of the block the worker is able to check if the latest message of the forward analysis satisfies the current post-condition. The distributed predicate analysis would check, whether $u(\phi_S) \wedge u(\psi)$ is satisfiable. For this check, the uninstantiate function u described in Section 4.2.1 comes in handy, as the most recent state of a variable is given by the variable instantiated with index 0. If this is not the case, we can abort the calcu-

lation as the error condition is proven unreachable via this block. We can decline *error conditions* in this case since the latest computed precondition for successors suffices to provoke a contradiction and so will any preciser precondition in the future. Since the post-condition of a successor ends on the final location of the current block, we can simply conjunct both formulas and perform a SAT-check. In general, the *proceed operator* should return *false* if the two conditions are not compatible. The defined *proceed operator* in Section 4.2.2 for the distributed predicate analysis satisfies this requirement with Case 2.2.

4.1.2 Decomposition of CFAs

Before we describe one possible approach to decompose a CFA, we have to define *merging locations* and *branching locations*.

Definition 12 (Merging Locations) A location $l_M \in L$ of a CFA (L, l_0, G) is called *merging location* if it has at least two entering edges, i.e., $|\{(l, *, l') \in G \mid l' = l_M\}| \geq 2$

Definition 13 (Branching Location) A location $l_D \in L$ of a CFA (L, l_0, G) is called *branching location* if it has at least two outgoing edges, i.e., $|\{(l, *, l') \in G \mid l = l_D\}| \geq 2$

Definition 14 (X-Location) A location l is called X-node or X-location if it is either a *branching location* or a *merging location*.

Note, that a node can be both, a *branching* and a *merging location*. We can get a real decomposition of a CFA if every block $B \in \mathcal{B}$ starts with a branching/merging node and ends with the next reachable branching/merging node if one follows the outgoing edge(s) from the start node.

Linear Decomposition

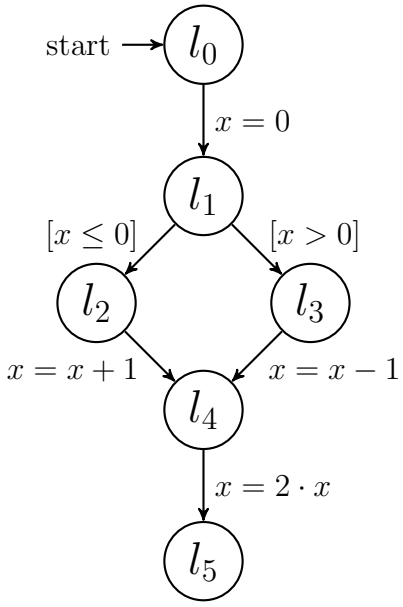
Algorithm 2 shows the procedure to obtain the desired decomposition. In line 1 we create our set $\mathcal{R}$ of known merging/branching nodes, called X-nodes from now on. The set $\mathcal{C}$ contains all covered X-nodes and $\mathcal{B}$ will store all found blocks. The following while-loop in line 4 calculates all *code blocks*. First, we remove the already covered X-nodes from $\mathcal{R}$ as a cyclic subgraph of the CFA may end on an already processed X-location. Afterwards, we add all remaining elements to $\mathcal{C}$ because they get covered in the upcoming for-loop. We now compute all paths to every directly reachable X-node starting at l_0. This means that a path does contain exactly two X-nodes, namely l_0

Algorithm 2: LinearDecomposition

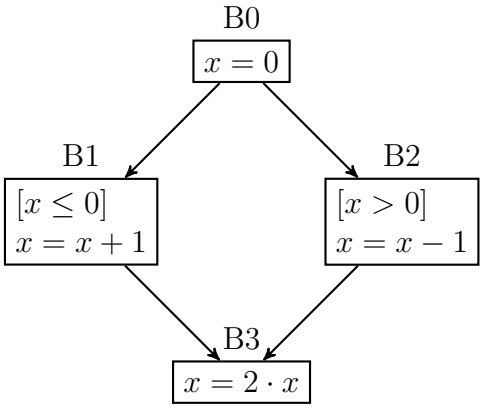
Input: (L, l_0, G): CFA
Output: $\mathcal{B}$: Set of blocks
Result: Decomposes the CFA to *code blocks* and stores them in a set

```
1  R = {l_0};                                  // merging/branching nodes
2  C = {};                                     // covered merging/branching nodes
3  B = {};                                     // found code blocks
4  while |R| ≠ 0 do
5      R = R \ C;
6      C = C ∪ R;
7      for l_0 ∈ R do
8          for p = (l_0, ..., l_n) ∈ allPathsToFollowingXNodes(l_0) do
9              L_B = ⋃_{l∈p} {l};
10             G_B = ⋃_{0≤i<n} {(l_i, *, l_{i+1})};
11             B = B ∪ {(L_B, l_0, l_n, G_B)};
12             R = R ∪ {l_n};
13 return B;
```

and l_n. In programs with loops l_n may equal l_0. Subsequently, we transform every found path to a *block node*. The *block node* consists of the locations L_B (all locations that are part of path p), the edges G_B (all edges between consecutive locations in path p), the start location l_0 and the final location l_n. All l_n are X-nodes and thus we add them to the set $\mathcal{R}$. Eventually, there are no X-nodes left in $\mathcal{R}$ and the calculation finishes by returning all *block nodes*. Sometimes, our approach benefits from less blocks as this reduces the number of messages sent between our actors. The decomposition presented in Algorithm 2 usually produces many *code blocks* resulting in many actors. The name LinearDecomposition is derived from the fact that no *code block* contains a branching (*linear blocks*). Figure 7 demonstrates how the linear decomposition works on an example CFA known from Section 3.1. We start at the entry node l_0. Now, we compute all direct paths to successive X-nodes. In this case, our only path p consists of one transition $(l_0, x = 0, l_1) \in G$. We add this path as a new block $B = (\{l_0, l_1\}, l_0, l_1, \{(l_0, x = 0, l_1)\})$ to $\mathcal{B}$ and l_1 is added to $\mathcal{R}$. Next, we calculate all paths from l_1 to successive X-nodes. This time we get two paths, $p_1 = ((l_1, [x \leq 0], l_2), (l_2, x = x + 1, l_4))$ and $p_2 = ((l_1, [x > 0], l_3), (l_3, x = x - 1, l_4))$. We add both paths as blocks to $\mathcal{B}$ and l_4 to $\mathcal{R}$. We repeat this until we cannot find paths anymore. Eventually, we reach location l_5 and stop. Finally, we obtain the four *linear blocks* B0,



(a) CFA



(b) Block nodes according to LinearDecompistion

Figure 7: Linear decomposition of a CFA

B1, B2 and B3. For the sake of simplicity, we do not draw the subgraph within a *code block* if it is linear (c.f. Figure 7 b)). The algorithm matches Definition 7 of a real decomposition.

Given Size Decomposition

As mentioned above, the LinearDecomposition algorithm produces many blocks. Later, we will see that every block needs one worker and thus an extra thread. Many blocks, i.e., workers need more resources. To encounter this issue, we introduce the GivenSizeDecomposition.

Algorithm 3 shows the procedure for the reduction of blocks $\mathcal{B}$ over a CFA (L, l_0, G) converging against the given number s. First, we run LinearDecomposition on the given CFA to get linear blocks. Afterwards, we store the initial location (l_0) and the final location (l_f) of each node in a separate set H. This enables us to create an entryPointMap in line 3 that maps every tuple of initial and final location in H to all blocks matching $(*, l_0, l_f, *)$, i.e., having the exact same entry points. Then, we enter the first for-loop. We iterate over all tuples containing the entry points and the corresponding set of blocks $\mathbb{B}$. We combine all blocks in $\mathbb{B}$ to one block by merging the locations and the edges. We exit the for-loop as soon as we reach the desired number of blocks s or there are no mergeable blocks left. If s is still less

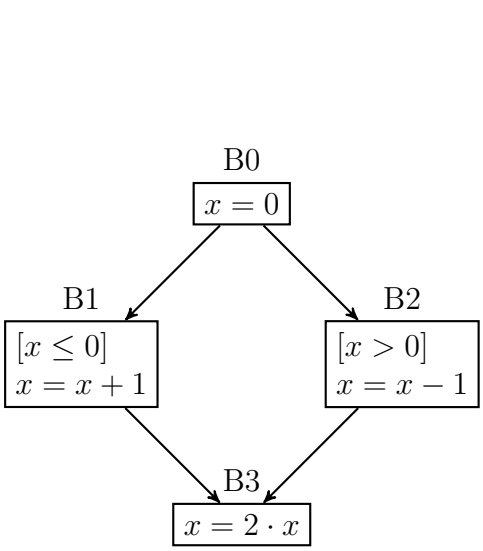
Algorithm 3: GivenSizeDecomposition

Input: (L, l_0, G): CFA
Input: s: desired number of *code blocks*
Output: $\mathcal{B}$: Set of blocks
Result: Decomposes the CFA to *code blocks* and stores them in a Block Tree

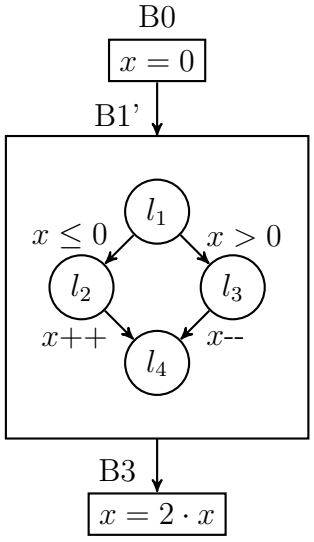
1 $\mathcal{B} = \text{LinearDecomposition}((L, l_0, G))$;
2 $H = \{(l_{B_0}, l_{B_f}) | B \in \mathcal{B}\}$;
3 entryPointMap $= \{((l_{B_0}, l_{B_f}), \{B | B = (L_B, l'_{B_0}, l'_{B_f}, G_B) \in \mathcal{B} \wedge (l_{B_0}, l_{B_f}) = (l'_{B_0}, l'_{B_f})\}) | h \in H\}$;
4 **for** $((l'_{B_0}, l'_{B_f}), \mathbb{B}) \in$ *entryPointMap* $\wedge\ s < |\mathcal{B}| \wedge |\mathbb{B}| > 1$ **do**
5 $\quad \mathcal{B} = \mathcal{B} \setminus \mathbb{B}$;
6 $\quad L_B = \bigcup_{(L_{B'}, *, *, *) \in \mathbb{B}} L_{B'}$;
7 $\quad G_B = \bigcup_{(*, *, *, G_{B'}) \in \mathbb{B}} G_{B'}$;
8 $\quad B' = (L_B, l'_{B_0}, l'_{B_f}, G_B)$;
9 $\quad \mathcal{B} = \mathcal{B} \cup \{B'\}$
10 **for** $B = (L_B, *, l_{B_f}, G_B) \in \mathcal{B} \wedge\ s < |\mathcal{B}|$ **do**
11 $\quad \mathbb{B} = succ(B)$; `// c.f. Definitions 3 and 4`
12 $\quad$ **if** $|\mathbb{B}| = 1$ **then**
$\quad\quad$ `/* Assume` $\mathbb{B} = \{B_S\}$ `*/`
13 $\quad\quad$ **if** $|pred(B_S)| = 1$ **then**
14 $\quad\quad\quad \mathcal{B} = \mathcal{B} \setminus (\mathbb{B} \cup \{B\})$;
15 $\quad\quad\quad \mathcal{B} = \mathcal{B} \cup \{(L_B \cup L'_B, l_{B_0}, l_{B_f}, L_G \cup L'_G) | (L'_B, *, *, L'_G) \in \mathbb{B}\}$;
16 **return** $\mathcal{B}$;

than the number of blocks in $\mathcal{B}$ we apply another merge strategy, starting at line 10. For every block with the final location l_f, we count the number of blocks with the initial location l_f. In case, there is exactly one block with the initial location l_f, we can merge both blocks if the second block has only one predecessor as they are direct successors (c.f. Figure 9b).

We let the GivenSizeDecomposition run on the CFA from before and set $s = 3$. As described above, blocks B1 and B2 have the same initial and final location and thus can be merged. Since, we found a decomposition having exactly 3 blocks the algorithm stops. The result can be seen in Figure 8. However, if we set $s = 2$ initially, the algorithm would still exit the first for-loop because there are no nodes left to merge horizontally, but it would enter the second for-loop as there are nodes that potentially can be merged

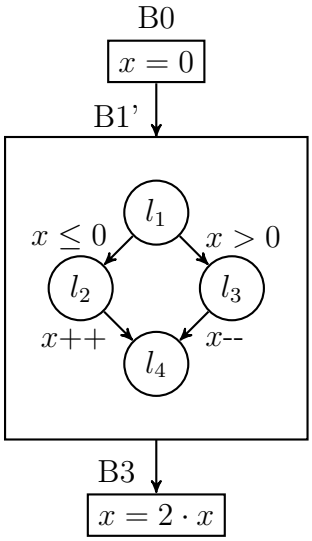


(a) Code blocks according to LinearDecomposition

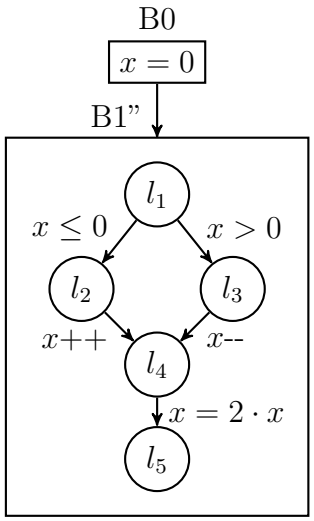


(b) Merging former blocks B1 and B2 into B1'

Figure 8: Result of GivenSizeDecomposition with $s = 3$



(a) Decomposition in 3 blocks



(b) Decomposition in 2 blocks

Figure 9: Result of GivenSizeDecomposition with $s = 2$

vertically. Consider block B1' as the current block in the for-loop in line 10. The final location of B1' equals l_4 and there is only one block, namely B3,

Algorithm 4: DcpaAlgorithm

Global: $B = (L_B, l_{B_0}, l_{B_f}, G_B)$: *code block*
Global: ($\mathbb{C}$, serialize, deserialize, proceed, combine): DCPA
Input: msg: Message
Output: Set of messages

1 shouldProceed, $\mathbb{M}$ = proceed(msg);
2 **if** $\neg$*shouldProceed* **then**
3 **return** $\mathbb{M}$;
4 e_0 = combine$_E$({deserialize(m) | $m \in \mathbb{M}$}); `// get initial state`
5 $R = \text{CPA}(\mathbb{C}, e_0, B)$; `// run CPA`
6 response = combine$_E$({e| $e \in R \setminus \{e_0\} \wedge location(e) = l_{B_f}$});
7 **return** {serialize(response)}; `// serialize knows type` τ

Result: The answer to msg

with the initial location l_4. Hence, we can merge them vertically. The result is depicted in Figure 9. Again, this algorithm satisfies our Definition 7 of a real decomposition. In some cases, Algorithm 3 fails to reach the desired number of blocks. It will merge all possible blocks and returns them instead.

If we cannot reach the desired number of blocks after the first run, we can repeatedly apply the GivenSizeDecomposition. We stop if the number of blocks does not change anymore.

4.1.3 Algorithm for Distributed Analysis

Algorithm 4 shows the basic concept of every distributed CPA. At the beginning, we check whether the DCPA lets us proceed by calling its *proceed operator*. It returns a Boolean value shouldProceed and a set of messages $\mathbb{M}$ (line 1). Now, we check whether shouldProceed equals *false* and return $\mathbb{M}$ if it is the case (lines 2 and 3). Otherwise, we deserialize all messages in $\mathbb{M}$ and combine them to one abstract state e with the help of the *combine operator* (line 4). Next, we run the normal CPA $\mathbb{C}$ on our block B with the initial abstract state e_0. The CPA returns the set of reached abstract states R. We extract all abstract states at the final location l_{B_f} of our block B. Since there is the chance that the final location equals the initial location, we have to remove e_0 from this set. Finally, we combine valid abstract states to one abstract state and return the serialized version of it as a singleton set. The initial messages of type BlockPostcondition comply with the concept of *large block encoding* [4] as either a sequence of statements or a choice of paths is generated. The latter is only possible with the GivenSizeDecomposi-

Table 4: Overview of responses of workers to certain message types

Worker	**BP**	**EC**	**ECU**	**R**	**E**
Analysis Worker	DCPA_F	DCPA_B	-	↯	↯
Fast Analysis Worker	DCPA_F	DCPA_B	-	↯	↯
Fault Loc. Worker	DCPA_F	DCPA_B + FL	-	↯	↯
Root Worker	-	DCPA_B	-	↯	↯
Result Worker	-	adapt counter	adapt counter	↯	↯
Timeout Worker	-	-	-	↯	↯
Visualization Worker	log	log	log	↯	↯

tion. Instead of rescheduling a new forward analysis, we could also conjunct the precondition of a worker with the results of the initial forward analysis.

4.1.4 Distributed Verification on Merged Blocks

In Section 4.1.2, we show an algorithm to merge blocks (GivenSizeDecomposition). This leads to blocks containing both, an error location and a reachable location at the block end. Blocks with this property send multiple messages. One message of type BlockPostCondition and one message of type ErrorCondition, originating from different locations in the block. It is also possible for such blocks to contain multiple unmergeable abstract states at the initial location of a block after a backward analysis. In such cases, we send one message of type ErrorCondition for each state instead of combining them. This ensures that we consider each possible error path separately.

4.1.5 Overview

Workers are entities of our actor model that process one message at a time. The response consists of a (potentially empty) set of messages. Table 4 shows a summary of how different types of workers react to certain message types (**Result**, **Error**, **BlockPostcondition**, **ErrorCondition**, **ErrorConditionUnreachable**). We use the ↯-symbol to show that a worker shuts down. Table 5 lists all worker types and depicts which message types can be broadcasted by a specific worker. Every worker is able to send messages of type Error. The verification result can only be determined by the *result* and *root worker*. The *timeout worker* schedules a result message (result: unknown) after a user-defined amount of time expires. All kinds of *analysis worker* (described in Section 4.1.6) can tell whether an *error con-*

Table 5: Overview of message types a worker can broadcast

Worker	**BP**	**EC**	**ECU**	**R**	**E**
Analysis Worker	✓	✓	✓	✗	✓
Fast Analysis Worker	✓	✓	✓	✗	✓
Fault Loc. Worker	✓	✓	✓	✗	✓
Root Worker	✓	✗	✓	✓	✓
Result Worker	✗	✗	✗	✓	✓
Timeout Worker	✗	✗	✗	✓	✓
Visualization Worker	✗	✗	✗	✗	✓

Algorithm 5: WorkerRoutine

```
1 while ¬finished do
2     broadcast(processMessage(awaitNextMessage()))
```

dition is unreachable. Additionally, they, as well as the *root worker*, issue a first BlockPostcondition message after the first forward analysis and react to messages of that type. *Root workers* will never receive a message of type BlockPostcondition as they do not have predecessors. Additionally, *analysis workers* are able to send messages of type ErrorCondition whenever they finish a backward analysis.

4.1.6 Worker Architecture

So far, we have discussed how to decompose a CFA and how actors can communicate over messages. Moreover, we know how distributed CPAs work. Now, we have all pieces to describe how our actors are functioning. Conceptually, the task of a worker is to transform a received message to a potentially empty set of answers that are messages, too. Hence, our actors are resembled by *workers* that execute an arbitrary algorithm triggered by a newly received message. Whenever a worker produces information worth broadcasting it sends a message as defined in Definition 8 to every other subscribed actor. Formally, each worker executes the code shown in Algorithm 5 and only implements processMessage separately. All workers run parallel. The method processMessage calculates a set of messages as answer to the message returned by awaitNextMessage. The method awaitNextMessage blocks the worker until a new message arrives. Eventually, the Boolean variable finished is set to *true* and the worker exits the loop. This section lists all available workers and explains how they transform received messages to new information for

other workers. If any worker runs into an unexpected error, it broadcasts a message with the type $t_\omega = \text{ERROR}$ causing all other workers to shutdown. The verification result equals UNKNOWN in this case.

Analysis Worker

Every *analysis worker* maintains two distinct analyses. A forward analysis DCPA_F and a backward analysis DCPA_B. They are called, depending on the type of the message returned from awaitNextMessage. *Analysis worker* operate on a block $B = (L_B, l_{B_0}, l_{B_f}, G')$ and manage the communication with other workers. Since workers do not start analyzing until they receive a message, we have to manually trigger the first forward analysis for every *analysis worker* directly after their creation. Afterwards, no manual interaction is required.

Forward Analysis Messages of the type $t_\omega = \text{BLOCKPOSTCONDITION}$ trigger a forward analysis DCPA_F if $l_{B_0} = l_{id}$, i.e., if the initial location of the block of the worker equals the target node of the message. As we already know, this is the task of the *proceed operator*. A forward analysis can either result in a new message of type $t_\omega = \text{BLOCKPOSTCONDITION}$ or in a new message of type $t_\omega = \text{ERRORCONDITION}$. The latter is caused by finding a reachable error location in block B. If no error location is reachable, we broadcast the latest combination of abstract states at l_{B_f} such that other analysis can update their assumptions about what is valid at their initial location of their *code block*. As mentioned before, workers operating on merged blocks may send multiple messages.

Backward Analysis Messages of the type $t_\omega = \text{BLOCKPOSTCONDITION}$ trigger a backward analysis DCPA_B if $l_{B_f} = l_{id}$, i.e., if the final location of the block of the worker equals the target node of the message. The backward analysis operates on B^T, the flipped version of B as described in Definition 10. If the *proceed operator* signals to continue, the worker broadcasts a new message of type $t_\omega = \text{ERRORCONDITION}$. Therefore, we broadcast a message for every abstract state at l_{B_0} since l_{B_0} is the final location of B^T. In case the *proceed operator* stops the backward analysis, the worker confirms the unreachability of the error condition by broadcasting a message of type ERRORCONDITIONUNREACHABLE.

Fast Analysis Worker

The number of messages grows with the number of workers and there is the chance that the queue of pending messages is large. To tackle the problem,

fast analysis workers immediately discard all uninteresting messages instead of processing them one by one and additionally, all messages of type $\tau =$ BlockPostcondition are processed as one abstract state instead of many different abstract states by using the deserialize and combine operators. Imagine, our queue contains n messages $\mathcal{M} = \bigcup_{0 \leq i < n} \{$(BlockPostcondition, *, *, $\{ID_{\mathbb{C}} : \phi_i\})\}$ then we issue a new analysis with the abstract state $combine_E(\{deserialize(m) | m \in \mathcal{M}\})$ instead of issuing n analyses with the respective abstract state representing any ϕ_i. The payload ρ maps an identifier $ID_{\mathbb{C}}$ for a CPA to the information ψ_i needed to deserialize a message to an abstract state of the abstract domain of $\mathbb{C}$. Other than that, this worker does not differ from the *analysis worker* described above.

Root Worker

The *root worker* only serves one purpose: to detect violations of the specification. Whenever the root worker receives a message m matching the pattern (ErrorCondition, *, 0, *) it checks whether proceed(deserialize (m)) returns true. In case, proceed wants to proceed, the root worker broadcasts the message $M_r =$ (Result, root, 0, {result: False}), telling the other workers that the input program is proven unsafe and that they can stop working. In other words, the current abstract state represents a feasible abstract state. Since it summarizes the path from the error location to the root, an error location is reachable. Hence, we can finish the analysis. Directly after its creation, the *root worker* sends a BlockPostcondition message representing $\top$ at location l_0 once.

Fault Localization Worker

The *fault localization worker* extends the *analysis worker*. Whenever the *analysis worker* wants to broadcast a message of type $\tau_w =$ ErrorCondition, it additionally triggers the fault localization (FL). Currently, *fault localization workers* execute the MaxSat algorithm [16, 17]. More details about the *fault localization worker* are covered in Section 4.3.

Result Worker

The *result worker* keeps track of all messages of type ErrorCondition and ErrorConditionUnreachable. Whenever the number of processed messages of type ErrorConditionUnreachble equals the number of workers that received a message of type ErrorCondition, this worker broadcasts $M_R =$ (Result, result, 0, {result: True}). All workers will shutdown as the program has been proven save. In case, a worker answers with a stronger

error condition, the *result worker* does not wait for an message of type ErrorConditionUnreachable anymore.

Timeout Worker

The *timeout worker* ignores every received message. Instead, directly after its creation, it starts a timer scheduling the message M_t = (Result, timeout, 0, {result: UNKNOWN}). After the timer expires all workers will shutdown whenever they process M_t.

Visualization Worker

The *visualization worker* logs every message to a file. An external Python program parses the file and transforms it into a HTML report including the block graph. Additionally, the report contains a table listing all messages from *analysis workers* sorted by their timestamp (time of creation).

4.2 Distributed Predicate CPA

4.2.1 Uninstantiating Path Formulas

Before we continue with the definition of the distributed predicate CPA, we have to define a useful function $u(\phi)$ that allows us to broadcast path formulas without additionally providing the SSA-map. This safes time whenever we deserialize a message and it reduces the size of messages. We explain the function with the help of an example. Consider the path formula

$$\phi \Leftrightarrow \overbrace{x^{\langle 0\rangle} = 5}^{\text{x = 5}} \wedge \overbrace{x^{\langle 1\rangle} = x^{\langle 0\rangle} + 1}^{\text{x++}} \wedge \overbrace{x^{\langle 2\rangle} = 2}^{\text{x = 2}} \wedge \overbrace{y^{\langle 0\rangle} = 0}^{\text{y = 0}} \wedge \overbrace{y^{\langle 0\rangle} \leq 0}^{\text{y<=0}}.$$

with the SSA-map

$$s = \{x : 2, y : 0\}.$$

We can query the highest SSA-index of a variable v by calling $s(v)$, e.g., $s(x) = 2$. First, u extracts all distinct variables. Variables with different SSA-indices are considered to be distinct. From the example above u extracts the following variables V where

$$V = \{x^{\langle 0\rangle}, x^{\langle 1\rangle}, x^{\langle 2\rangle}, y^{\langle 0\rangle}\}.$$

For every variable $v \in V$ we call the function

$$f(v^{\langle n\rangle}) = \begin{cases} v^{\langle 0\rangle} & \text{if } n = s(v) \\ v.uid & \text{else} \end{cases}$$

where *uid* is an unique identifier that never repeats. In the simplest case, we can define the *uid* over the set of natural numbers $\mathbb{N}$. Whenever we need a *uid*, we remove one element $id \in \mathbb{N}$ from $\mathbb{N}$ such that the next call cannot give us the same number again. In our example $f(x^{\langle 0 \rangle}) = x.0, f(x^{\langle 1 \rangle}) = x.1, f(x^{\langle 2 \rangle}) = x^{\langle 0 \rangle}$ and $f(y^{\langle 0 \rangle}) = y^{\langle 0 \rangle}$ holds. In the last step, we replace every variable $v \in V$ in ϕ with $f(v)$. We obtain

$$u(\phi) \Leftrightarrow x.0 = 5 \wedge x.1 = x.0 + 1 \wedge x^{\langle 0 \rangle} = 2 \wedge y^{\langle 0 \rangle} = 0 \wedge y^{\langle 0 \rangle} \leq 0.$$

Note, that the renaming does not change the semantics of the formula. We still express the exact same formula. We achieved that the most recent variable has the SSA-index 0. The formula $u(\phi)$ is ready to be broadcasted. The function u works for the forward as well as for the backward analysis. The example above showcased the transformation of a formula produced by a forward analysis. Imagine the backward analysis traverses the same sequence of statements in reversed order, then

$$\phi \Leftrightarrow \overbrace{y^{\langle 0 \rangle} \leq 0}^{\text{y<=0}} \wedge \overbrace{y^{\langle 0 \rangle} = 0}^{\text{y = 0}} \wedge \overbrace{x^{\langle 0 \rangle} = 2}^{\text{x = 2}} \wedge \overbrace{x^{\langle 1 \rangle} = x^{\langle 2 \rangle} + 1}^{\text{x++}} \wedge \overbrace{x^{\langle 2 \rangle} = 5}^{\text{x = 5}}.$$

Remember that the backward analysis increases the SSA-index of variables after it assigned the value to it. Thus, our SSA-map s has the following values:

$$s = \{x : 3, y : 1\}.$$

Our set of variables $V = \{x^{\langle 0 \rangle}, x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, y^{\langle 0 \rangle}\}$ stays unchanged but the SSA-map is different. If we now call the function f for every variable we get $f(x^{\langle 0 \rangle}) = x.2, f(x^{\langle 1 \rangle}) = x.3, f(x^{\langle 2 \rangle}) = x.4$ and $f(y^{\langle 0 \rangle}) = y.5$. Recall, that f does not return any *uid* twice. Our uninstantiated path formula equals

$$u(\phi) \Leftrightarrow y.5 \leq 0 \wedge y.5 = 0 \wedge x.2 = 2 \wedge x.1 = x.2 + 1 \wedge x.2 = 5.$$

The uninstantiated formula does not contain any information about the most recent variable ($x^{\langle 0 \rangle}$ or $y^{\langle 0 \rangle}$) because we do not need that information. It does not matter what the values of x and y were in other blocks since this blocks overrides both values with new ones. Note, that u is idempotent, i.e., $u(u(\ldots(\phi))) = u(\phi)$. We call the formula *uninstantiated* since the variables either have no SSA-index or the minimal SSA-index of 0. We use a "." to separate the variables from the SSA-index since no variable in the programming language C can have a dot in its name. Thus, the same variable name cannot exist elsewhere.

4.2.2 Definition of Distributed Predicate Analysis

Currently, only one distributed CPA is implemented. In this section, we give an implementation for every operator to extend the *predicate analysis* to a *distributed predicate analysis* running on the *code block* $B = (L_B, l_{B_0}, l_{B_f}, G_B)$. For now, the distributed predicate analysis does not support pointer arithmetic, abstraction and CEGAR. We solely use the predicate analysis to build the path formulas. However, the mentioned features can be integrated.

Serialize Operator

To serialize an abstract state $e = (*, *, *, \phi)$, we simply put the string representation φ of ϕ as value, together with the keyword *predicateCPA* into ρ.

$$\text{serialize}\,(e) = (\tau_\omega, B, l_{B_f}, \{\text{predicateCPA} : \varphi, \dots\})$$

The type τ_ω is set differently based on the context we are in. A backwards analysis sets $\tau_\omega = \textsc{ErrorCondition}$ while a forward analysis puts $\tau_\omega = \textsc{BlockPostcondition}$ instead.

Deserialize Operator

The *deserialize operator* deserializes the payload ρ of a given message $m = (*, *, *, \{\dots\})$ containing the key-value pair {predicateCPA: φ} where φ resembles a Boolean formula in string representation. We parse φ to an instance of a Boolean formula and calculate the SSA-indices. The Boolean formula and the SSA-map are put together to a path formula ϕ. This path formula is used to restore the abstract state.

$$\text{deserialize}\,((*, *, *, \{\text{predicateCPA} : \varphi, \dots\})) = (l_{B_0}, \mathit{true}, l_{B_0}, \text{parse}(\varphi)) = e$$

In case, there is no such key-value pair, we return $\top$.

Combine Operator

Let ϕ_1 and ϕ_2 be two path formulas of the abstract states e_1 and e_2, then $u(\phi_1) \vee u(\phi_2) = \phi_3$ is the path formula of the combined abstract predicate state e_3.

$$\text{combine}(e_1, e_2) = \text{combine}((l, *, *, \phi_1,), (l, *, *, \phi_2)) = (l, *, *, \phi_3) = e_3$$

Note, that this weakens the abstract states as the formula now represents two paths but only one has to be crucial for proving a program incorrect. Since we uninstantiate the formulas in advance, the SSA-indices are aligned. The most recent variables have the SSA-index 0.

Proceed Operator

Let $f_m = (\text{BLOCKPOSTCONDITION}, *, l_{B_f}, \{\text{predicateAnalysis: } \varphi_f, \dots\})$ be the last message with $\tau = \text{BLOCKPOSTCONDITION}$ that the *serialize operator* produced and $\mathbb{F}$ the set of all received messages of type BLOCKPOSTCONDITION, then the *proceed operator* for a new message $m = (\tau, id, l_{id}, \rho)$ is defined as follows:

Case 1: τ = BlockPostcondition

Case 1.1: $l_{id} = l_{B_0}$ $\mathbb{F}$ is complete if the *proceed operator* contains one BLOCKPOSTCONDITION message from each predecessor. New messages from a predecessors that already have sent a message are replaced as they are stronger. Hence, our new $\mathbb{F} = \{n | n = (*, n_{id}, *, *) \in \mathbb{F} \wedge n_{id} \neq id\} \cup \{m\}$.

Case 1.1.1: $\mathbb{F}$ complete The *proceed operator* returns

$$\text{proceed(m)} = (\mathit{true}, \mathbb{F}).$$

.

Case 1.1.2: $\mathbb{F}$ incomplete The *proceed operator* returns

$$\text{proceed(m)} = (\mathit{true}, \{\}).$$

Case 1.2: $l_{id} \neq l_{B_0}$ The *proceed operator* returns

$$\text{proceed(m)} = (\mathit{false}, \{\}).$$

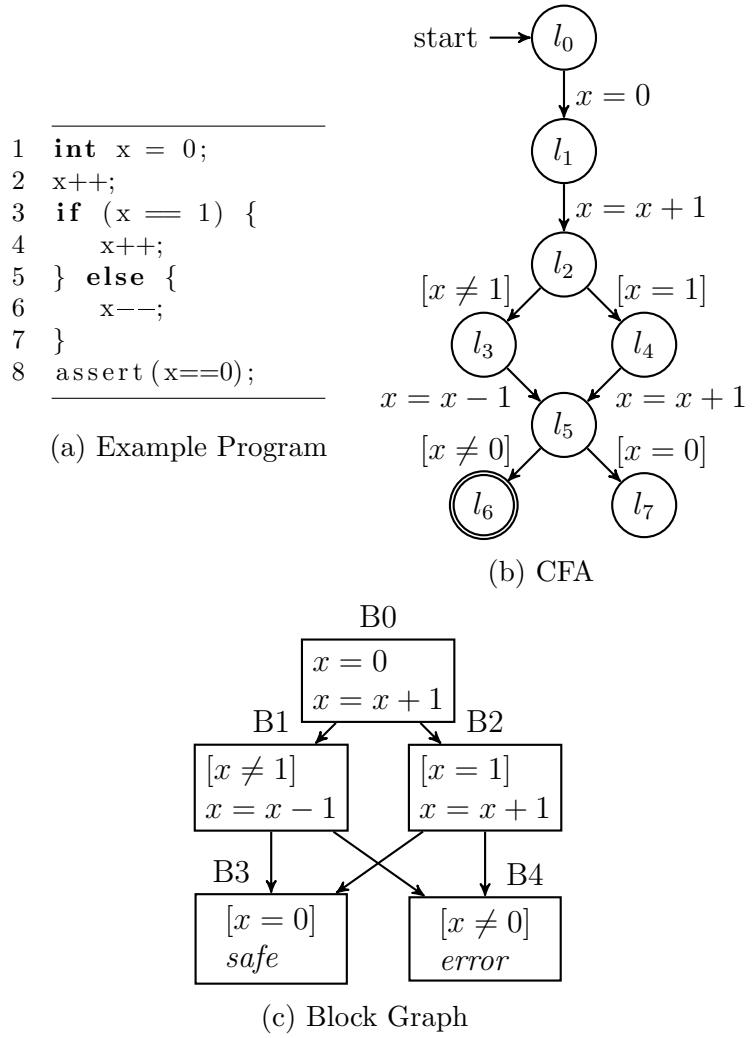
Case 2: τ = ErrorCondition If m has type $\tau = \text{BLOCKPOSTCONDITION}$, then check whether $\phi = u(\phi_1) \wedge u(\psi_f)$ is satisfiable where ϕ_1 is defined over the abstract state $e_1 = (*, *, *, \phi_1) = \text{deserialize}(m)$ encoded in the message m. The formula ψ_f represents the latest path formula computed by the forward analysis.

Case 2.1: ϕ is satisfiable The *proceed operator* returns

$$\text{proceed(m)} = (\mathit{true}, \{m\}).$$

Case 2.2: ϕ is unsatisfiable The *proceed operator* returns

$$\text{proceed(m)} = (\mathit{false}, \{(\text{ERRORCONDITIONUNREACHABLE}, \text{B}, *, *)\}).$$



(a) Example Program

(b) CFA

(c) Block Graph

Figure 10: A program represented as CFA and Block Graph

Case 3: Remaining types The *proceed operator* returns

$$\text{proceed(m)} = (\mathit{false}, \{\}).$$

4.2.3 Distributed Predicate Analysis by Example

We already illustrated the conversion of programs to a CFA and subsequently to *code blocks*. Figure 10 shows this for the given program in Figure 10a. Now we create a worker for every block. As discussed earlier, every block has its own precondition and its own postcondition. Initially all of them equal $\top$. To simplify the illustration, we assume that all workers take the exact same

Table 6: Initial messages of all workers

	W_0	W_1	W_2	W_3	W_4
ψ	$\top$	$\top$	$\top$	$\top$	$\top$
ϕ	$\top$	$\top$	$\top$	$\top$	$\top$
τ	BP	BP	BP	BP	EC
id	W_0	W_1	W_2	W_3	W_4
l_{id}	2	5	5	6	5
ρ	$x.0 = 0 \wedge$ $x^{\langle 0 \rangle} = x.0 + 1$	$x.1 \neq 1 \wedge$ $x^{\langle 0 \rangle} = x.1 - 1$	$x.2 = 1 \wedge$ $x^{\langle 0 \rangle} = x.2 + 1$	$x^{\langle 0 \rangle} = 0$	$x^{\langle 0 \rangle} \neq 0$

time to process one message, no matter what message they process. Irrelevant messages take no time at all. All workers run the distributed predicate analysis for the forward and backward analysis. Right after the creation, every worker runs a forward analysis first. We assume that W_i operates on B_i. Since blocks 0, 1, 2 and 3 do not have error locations in them, the workers broadcast a message to update the precondition of successive workers. However, block 4 contains an error location and thus starts a backward analysis broadcasting an error condition message targeting location node 5. At this point, every worker has five messages to process (Table 6). Worker W_0 discards all messages and is done for now as no message contains the location 0 with type BP and no message contains the location id 2 with the type EC. With the same reasoning, worker W_1 only processes the messages from W_0 and W_4. The precondition updates to $\psi_1 \Leftrightarrow x.0 = 0 \wedge x^{\langle 0 \rangle} = x.0 + 1$ and with that update the worker schedules a new forward analysis. Afterwards, the worker updates its post-condition because of the message from W_4 and runs a backward analysis as we will see later. The same applies to worker W_2 because it has the same initial and final location as W_1. W_3 receives two valuable messages. First, the message from W_1 updates the precondition of W_3 to $\psi_3 \Leftrightarrow x.1 \neq 1 \wedge x^{\langle 0 \rangle} = x.1 - 1$ but it does not schedule a new forward analysis because it did not receive updates from all predecessors. If we would start the forward analysis now we would under approximate as we only cover one path. We have to wait until the message from W_2 arrives. Luckily, this is the next message in the queue. W_3 now updates its precondition to $\psi_3 \Leftrightarrow (x.1 \neq 1 \wedge x^{\langle 0 \rangle} = x.1 - 1) \vee (x.2 = 1 \wedge x^{\langle 0 \rangle} = x.2 + 1)$ and runs a forward analysis. To reduce the size of the formula, we simplify it to the equivalent formula $\psi_3 \Leftrightarrow x^{\langle 0 \rangle} \neq 0 \vee x^{\langle 0 \rangle} = 2$. In reality, the worker will not simplify the formulas. We once again benefit from the uninstantiation with u as the left and the right hand side of the disjunction correctly talk about the same $x^{\langle 0 \rangle}$

Table 7: First updates to the preconditions

	W_1	W_2	W_3	W_4
ψ	$x^{\langle 0 \rangle} = 1$	$x^{\langle 0 \rangle} = 1$	$x^{\langle 0 \rangle} = 2 \vee$ $x^{\langle 0 \rangle} \neq 0$	$x^{\langle 0 \rangle} = 2 \vee$ $x^{\langle 0 \rangle} \neq 0$
ϕ	$\top$	$\top$	$\top$	$\top$
τ	BP	BP	BP	EC
id	W_1	W_2	W_3	W_4
l_{id}	5	5	6	5
ρ	*false*	$x.3 = 1 \wedge$ $x.3 = 1 \wedge$ $x^{\langle 0 \rangle} = x.3 + 1$	$(x^{\langle 0 \rangle} = 2 \vee$ $x^{\langle 0 \rangle} \neq 0) \wedge$ $x^{\langle 0 \rangle} = 0$	$x \neq 0$

Table 8: Simplified preconditions for every workers

	W_0	W_1	W_2	W_3	W_4
ψ	$\top$	$x^{\langle 0 \rangle} = 1$	$x^{\langle 0 \rangle} = 1$	$x^{\langle 0 \rangle} = 2$	$x^{\langle 0 \rangle} = 2$
ψ_W	$x^{\langle 0 \rangle} = 1$	*false*	$x^{\langle 0 \rangle} = 2$	*false*	none
ϕ	$\top$	$\top$	$\top$	$\top$	$\top$

without further adaptions. At this location, x either equals 2 or is unequal to 0 and thus we need to talk about the same variable. Since W_4 has the same initial location as W_3, W_4 has the same precondition and also schedules a forward analysis. Table 7 shows the result of the forward analyses triggered by the above mentioned messages of type BP. Since W_0 discarded all messages, it does not appear in Table 7. W_1 calculates a new precondition for its successors with the value *false* meaning that this path is not reachable at all. From now on, the distributed analysis concentrates on paths without W_1. No error will ever contain edges from G_{B_1}. All other workers calculate the new updates as before. W_4 can still reach the error and thus broadcasts the exact same message as before again. W_3 and W_4 update their preconditions to $\mathit{false} \vee x.3 = 1 \ \wedge x.3 = 1 \ \wedge x^{\langle 0 \rangle} = x.3 + 1 \Leftrightarrow x^{\langle 0 \rangle} = 2$ and automatically do not consider paths over W_1 anymore. The current state is depicted in Table 8. We add the row ψ_W denoting the already simplified path formula of the latest message of type $\tau = \text{BP}$ sent by the respective worker. W_1 and W_3 cannot reach their block end and thus send *false*. Messages of type $\tau = \text{BP}$ of worker W_3 are ignored by all other workers. We will now talk about all pending ErrorCondition-messages with the current state of Table 8. The first message of this type was sent by W_4. Hence, workers W_1 and W_2 have to

handle it now. They use the *proceed operator* for that. The *proceed operator* of the distributed predicate analysis has access to both, the latest result of the own forward analysis ψ_{W_i} and the error condition ϕ sent by the successor. For block W_1 the *proceed operator* enters case 2.2 (c.f., Section 4.2.2) and checks whether $\psi_{W_1} \wedge \phi$ is satisfiable. If we put in the known values for them, we get the formula $\mathit{false} \wedge x \neq 0 \Leftrightarrow \mathit{false}$ which is unsatisfiable. Thus, the *proceed operator* returns (false, {(ErrorConditionUnreachable, ...)}) and the backwards analysis will not be triggered. Worker W_2 on the other hand, checks if $x^{\langle 0 \rangle} = 2 \wedge x^{\langle 0 \rangle} \neq 0$ is satisfiable and its *proceed operator* returns (true, $\{m\}$) where m is the message from W_4 (case 2.1 in Section 4.2.2). W_2 has to enter the backward analysis and computes the following post-condition for its predecessors $\phi_{W_2} \Leftrightarrow x^{\langle 0 \rangle} \neq 0 \wedge x^{\langle 0 \rangle} = x^{\langle 1 \rangle} + 1 \wedge x^{\langle 1 \rangle} = 1$. W_2 broadcasts the uninstantiated version $u(\phi_{W_2}) \Leftrightarrow x.4 \neq 0 \wedge x.4 = x^{\langle 0 \rangle} + 1 \wedge x^{\langle 0 \rangle} = 1$. As the only predecessors of W_2, W_0 picks up the message. Since $x.4 \neq 0 \wedge x.4 = x^{\langle 0 \rangle} + 1 \wedge x^{\langle 0 \rangle} = 1 \wedge x^{\langle 0 \rangle} = 1$ is satisfiable, W_0 enters the backward analysis and broadcasts an error condition message with the payload $x.4 \neq 0 \wedge x.4 = x.5 + 1 \wedge x.5 = 1 \wedge x.5 = x.6 + 1 \wedge x.6 = 0$. There is no instantiated x left, as the block would overwrite any results for x from predecessor blocks immediately with the statement $x = 0$.

Until now, we completely ignored the *result worker* and the *root worker*. The *result worker* tracks whether workers deny messages of type $\tau =$ ErrorCondition. If no worker has a ErrorCondition message left to process and the *root worker* does not report a violation, the *result worker* proves the program safe by broadcasting the corresponding message. However, in our example here, the *root worker* does report a violation. Remember, that the root worker only processes ErrorCondition messages with $l_{id} = 0$ which is the case in the latest message from W_0. The *root worker* calls its *proceed operator* with its precondition (always $\top$). In this example the *proceed operator* checks $\mathit{true} \wedge x.4 \neq 0 \wedge x.4 = x.5 + 1 \wedge x.5 = 1 \wedge x.5 = x.6 + 1 \wedge x.6 = 0 \Leftrightarrow x.4 \neq 0 \wedge x.4 = x.5 + 1 \wedge x.5 = 1 \Leftrightarrow x.4 \neq 0 \wedge x.4 = 2 \Leftrightarrow \mathit{true}$ which is satisfiable. Whenever the *root worker* receives a satisfiable error condition, the program is proven unsafe.

4.3 Distributed Fault Localization

To show that our approach is easily extendable to other concepts, we implement a distributed fault localization algorithm with the help of the distributed predicate analysis. In general, fault localization tries to find edges in the CFA that are especially error-prone and are most likely to cause the program to fail. In this section, we explain the MaxSat [16, 17] algorithm

and describe how it profits from the distributed approach.

4.3.1 Maximum Satisfiability Algorithm

The maximum satisfiability algorithm [16] (MaxSat) takes three inputs. A precondition ψ, a post-condition ϕ and a trace θ. In our case, ψ always equals the initial variable assignment and ϕ equals the condition leading to the error. The trace resembles a feasible path of the CFA that satisfies ϕ if ψ holds at the initial location of path θ. Consider Figure 10a once again. The initial variable assignment for the only variable x equals 0. Hence, our precondition can be written as $\psi \Leftrightarrow x = 0$. The negation of the assert statement in line 8 will be our post-condition $\phi \Leftrightarrow x \neq 0$ as we do not want x to equal any other value than 0. In the previous section, we have proven the program unsafe. Thus, we find a feasible path from line 1 to line 8 violating the assertion. The path $\theta = ((l_1, x = x + 1, l_2), (l_2, [x = 1], l_4), (l_4, x = x + 1, l_5))$ satisfies our post-condition if $x = 0$ initially. MaxSat operates on a so-called *trace formula* (TF).

Definition 15 (Trace Formula) The trace formula $\mathrm{TF}(\theta)$ equals the path formula of path θ in conjunction with the precondition and the negation of the post-condition.

In our case, the trace formula of θ equals

$$\mathrm{TF}(\theta) \Leftrightarrow \psi^{\langle 0 \rangle} \wedge x^{\langle 1 \rangle} = x^{\langle 0 \rangle} + 1 \wedge x^{\langle 1 \rangle} = 1 \wedge x^{\langle 2 \rangle} = x^{\langle 1 \rangle} + 1 \wedge \neg \phi^{\langle s \rangle}.$$

For the fault localization algorithm to work properly, $\mathrm{TF}(\theta)$ has to be unsatisfiable. With $\psi^{\langle 0 \rangle}$, we instantiate every variable in ψ with the SSA-index 0. Analogously, we instantiate every variable of $\phi^{\langle s \rangle}$ with the current maximal SSA-index of every variable in the SSA-map s. The trace formula for Figure 10a reads

$$\mathrm{TF}(\theta) \Leftrightarrow x^{\langle 0 \rangle} = 0 \wedge x^{\langle 1 \rangle} = x^{\langle 0 \rangle} + 1 \wedge x^{\langle 1 \rangle} = 1 \wedge x^{\langle 2 \rangle} = x^{\langle 1 \rangle} + 1 \wedge \neg (x^{\langle 2 \rangle} \neq 0).$$

The core idea of this fault localization technique is to find a subset $\theta_{\subseteq}$ of the path θ such that $\mathrm{TF}(\theta_{\subseteq})$ is satisfiable again. The complement of that set resembles the set of error-prone statements. For the program in Figure 10a, we start with an empty set $\mathbb{S}$. The function pf computes the path formula of θ. We keep adding edges to the set until $\psi^{\langle 0 \rangle} \wedge pf(\mathbb{S}) \wedge \neg \phi^{\langle s \rangle}$ turns unsatisfiable again. In this case, we remove the previously added edge and try the remaining edges. Naively speaking, we check every possible subset of the path θ and check whether the formula is satisfiable. The result of the algorithm is the complement of the biggest satisfying set. Without

Table 9: The fault localization algorithm checks every subset for satsifiability.

subsets	TF	satisfiable?
$\{\}$	$x^{\langle 0\rangle} = 0 \wedge x^{\langle 2\rangle} = 0$	✓
$\{x^{\langle 1\rangle} = x^{\langle 0\rangle} + 1\}$	$x^{\langle 0\rangle} = 0 \wedge$ $x^{\langle 1\rangle} = x^{\langle 0\rangle} + 1 \wedge$ $x^{\langle 2\rangle} = 0$	✓
$\{x^{\langle 1\rangle} = 1\}$	$x^{\langle 0\rangle} = 0 \wedge$ $x^{\langle 1\rangle} = 1 \wedge$ $x^{\langle 2\rangle} = 0$	✓
$\{x^{\langle 2\rangle} = x^{\langle 1\rangle} + 1\}$	$x^{\langle 0\rangle} = 0 \wedge$ $x^{\langle 2\rangle} = x^{\langle 1\rangle} + 1 \wedge$ $x^{\langle 2\rangle} = 0$	✓
$\{x^{\langle 1\rangle} = x^{\langle 0\rangle} + 1,$ $x^{\langle 1\rangle} = 1\}$	$x^{\langle 0\rangle} = 0 \wedge$ $x^{\langle 1\rangle} = x^{\langle 0\rangle} + 1 \wedge$ $x^{\langle 1\rangle} = 1 \wedge$ $x^{\langle 2\rangle} = 0$	✓
$\{x^{\langle 1\rangle} = x^{\langle 0\rangle} + 1,$ $x^{\langle 2\rangle} = x^{\langle 1\rangle} + 1\}$	$x^{\langle 0\rangle} = 0 \wedge$ $x^{\langle 1\rangle} = x^{\langle 0\rangle} + 1 \wedge$ $x^{\langle 2\rangle} = x^{\langle 1\rangle} + 1 \wedge$ $x^{\langle 2\rangle} = 0$	✗
$\{x^{\langle 1\rangle} = 1,$ $x^{\langle 2\rangle} = x^{\langle 1\rangle} + 1\}$	$x^{\langle 0\rangle} = 0 \wedge$ $x^{\langle 1\rangle} = 1 \wedge$ $x^{\langle 2\rangle} = x^{\langle 1\rangle} + 1 \wedge$ $x^{\langle 2\rangle} = 0$	✗

any optimizations, we have to make $2^{|\theta|}$ satisfiability checks (in our case 8). In prior work [17], we show that we can find significant improvements to reduce the number of SAT-checks. However, in the worst-case, we still have to check all subsets. Table 9 shows all subsets and indicates whether the resulting trace formula together with the pre- and post-conditions is satisfiable. We do not have to check the complete set as we already know that it would remain unsatisfiable by construction. The algorithm takes the subset of maximal size where the trace formula is still satisfiable and computes the complement. Here, we obtain $\{x^{\langle 2\rangle} = x^{\langle 1\rangle} + 1\}$ as complement

of $\{x^{\langle 1\rangle} = x^{\langle 0\rangle} + 1, x^{\langle 1\rangle} = 1\}$. Hence, the algorithm suggest to modify the statement $x^{\langle 2\rangle} = x^{\langle 1\rangle} + 1$ to fix the bug. In this synthetic task, we can fix the bug by replacing $+1$ with -2. We will now show the distributed fault localization algorithm.

4.3.2 Distributed MaxSat Algorithm

The distributed MaxSat algorithm (DMaxSat) uses the exact same algorithm as explained above for every block. We do not need to make any adaptions. The only thing remaining is to compute ψ, ϕ and lastly TF(θ). Currently, there is only one limitation: we have to use the LinearDecomposition algorithm to decompose the CFA. The advantage lies in the linearity of the blocks. Within the blocks no branchings exist and thus there is always only one path that can directly be used as θ. *Fault localization workers* trigger the MaxSat algorithm after every backward analysis. We can set θ to the path starting at the final location of the block of the worker and ending at the initial location of the block (backward analysis). The linearity ensures that there is only one such path. The post-condition ϕ for the fault localization equals the received *error condition*. To work properly, we have to make a small adaption to the post-condition. We describe the adaption later. Since the blocks might not know about the initial variable assignment yet, we have to compute a possible assignment by querying a model of the formula $u(\phi) \wedge pf(\theta)$. A model of a path formula assigns a possible concrete value to every variable in the formula such that replacing the variables with the respective value simplifies it to *true*. If no model is available, the error cannot be reached via the current block and the fault localization is not required. At the end, we only print the faulty locations of blocks that operate on the actual error path. The model does not require to find the true variable assignment because if the error is indeed reachable, the model automatically equals one concrete example a valid abstract state on this location would cover.

To illustrate the distributed fault localization, we run it exemplary on W_2 (c.f. Figure 10). W_2 receives the error condition $x^{\langle 0\rangle} \neq 0$ from W_4. The backward analysis computes the new error condition (before the uninstantiation) $x^{\langle 0\rangle} \neq 0 \wedge x^{\langle 0\rangle} = x^{\langle 1\rangle} + 1 \wedge x^{\langle 1\rangle} = 1$ which exactly equals the desired formula $\phi \wedge pf(\theta)$ were $\theta = ((l_5, x = x + 1, l_4), (l_4, [x = 1], l_2))$. We ask a solver to compute the model $\{x^{\langle 1\rangle} : 1, x^{\langle 0\rangle} : 2\}$. Because of the backward analysis, the initial variable assignment $\psi \Leftrightarrow x^{\langle 1\rangle} = 1$ consists of the values of the variables with the highest SSA-index. We feed ψ, θ and $t(\phi)$ to the MaxSat algorithm. The function t returns the *error condition* ϕ but negates the initial post-condition found by the first fault localization. We assume that the

Table 10: W_2 operates on a block with two statements. We check all subsets together with the pre- and postcondition for satisfiability and return the complement of the biggest unsatisfiable subset (second row).

subsets	TF	satisfiable?
$\{\}$	$x^{\langle 1\rangle} = 1 \wedge x^{\langle 0\rangle} = 0$	✓
$\{x^{\langle 0\rangle} = x^{\langle 1\rangle} + 1\}$	$x^{\langle 1\rangle} = 1 \wedge x^{\langle 0\rangle} = x^{\langle 1\rangle} + 1 \wedge x^{\langle 0\rangle} = 0$	✗
$\{x^{\langle 1\rangle} = 1\}$	$x^{\langle 1\rangle} = 1 \wedge x^{\langle 1\rangle} = 1 \wedge x^{\langle 0\rangle} = 0$	✓

initial post-condition for fault localization always equals the path formula of the assume edge closest to the actual error location.

Definition 16 Let $\phi \Leftrightarrow \bigwedge_{i=0}^{n} \phi_i$ be a formula with n conjunctions, then $t(\phi) \Leftrightarrow \bigwedge_{i=0}^{n} m(\phi_i)$ and $m(\phi) = \begin{cases} \neg\phi & \text{if } \phi \text{ equals the first post-condition} \\ \phi & \text{else} \end{cases}$

Table 10 shows the result. The complement of the maximum satisfiable set equals $\{x^{\langle 0\rangle} = x^{\langle 1\rangle}+1\}$ and that is the exact same location as computed before without the distributed approach. We need function t because our results turn invalid if the error condition contains non-assume edges. Suppose, we would just take the normal negation $x^{\langle 0\rangle} \neq x^{\langle 1\rangle} + 1 \vee x^{\langle 0\rangle} > 1$ of an error condition $x^{\langle 0\rangle} = x^{\langle 1\rangle} + 1 \wedge x^{\langle 0\rangle} \leq 1$ where the first statement is an assignment. Then, our post-condition becomes weaker and the actual reason of the bug disappears. The statement $x^{\langle 0\rangle} = x^{\langle 1\rangle} + 1$ is an integral element of the error trace. We would lose the information that $x^{\langle 0\rangle}$ depends on $x^{\langle 1\rangle}$ if we just apply normal negation. In our implementation, workers do not know what kind of statements are assumes as the path formula gives no hints about that fact ($x = 0$ and $[x = 0]$ both become $x^{\langle 0\rangle} = 0$). To keep track of the initial post-condition, the first worker adds it to the payload of a message. Subsequently, other workers copy it into their messages.

4.3.3 Run-Time Comparison

As already mentioned, the worst case runtime in terms of number of satisfiability checks of MaxSat is 2^n where n resembles the length of the error path θ. However, we have not covered the run-time of the distributed approach. For the sake of simplicity, we assume that our decomposition algorithm divides our program in m equally sized blocks. Every block now covers n/m locations of the error path and executes DMaxSat on that part leaving us with a total of $m \cdot 2^{n/m}$ satisfiability checks in the worst case. The total follows

Table 11: Run-time comparison with $n = 12$.

Algorithm	$m = 3$	$m = 4$	$m = 6$
MaxSat (2^n)	4096	4096	4096
DMaxSat ($m2^{n/m}$)	48	32	24

from the fact that m workers run fault localization on a path with approximately n/m transitions. If we now compare the two numbers, we see that there is a massive difference. With the assumptions that $2 \leq m \leq n$, i.e., the error path consists of at least two statements, we can prove our point.

$$\begin{aligned} 2^n &\geq m2^{n/m} && \mid \log_2(*) \\ n &\geq \log_2(m) + n/m && \mid \cdot\, m \\ mn &\geq m\log_2(m) + n && \mid -\, n \\ (m-1)n &\geq m\log_2(m) && \mid :\, m \\ (m-1)n/m &\geq \log_2(m) \end{aligned}$$

After the simplification we are left with three cases.

Case 1: Both variables have the value 2. We put the values in the above equation and get:

$$m = n = 2 \Rightarrow 1 \geq \log_2(2) = 1$$

Case 2: Both variables have the same value and they cancel from the fraction $(m-1)n/m = (m-1)$.

$$m = n \neq 2 \Rightarrow m - 1 \geq \log_2(m)$$

The statement is true as linear functions grow faster than logarithmic functions.

Case 3: In case of $m \leq n$, we know that $c = n/m \geq 1$. Thus, we can write

$$m \leq n \Rightarrow (m-1)n/m = c \cdot (m-1) \geq (m-1) \geq \log_2(m)$$

□

Table 11 shows the significant reduction of solver calls by approximately 99% in the worst case for $n = 12$ and $m \in \{3, 4, 6\}$. Once again, n is the length of θ and m the number of workers.

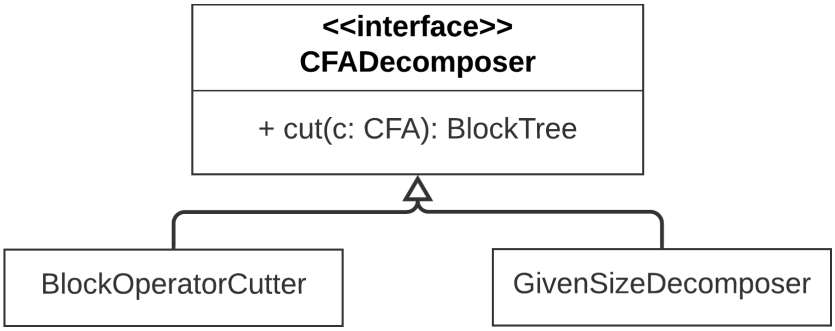


Figure 11: UML diagram of available decomposers

5 Implementation

This chapter covers the implementation details for our approach. We implement our work in CPAchecker[1] [7], a framework for formal verification. CPAchecker is implemented in Java.

5.1 Distributed Framework

5.1.1 Decomposition

In Section 4.1.2, we showcase two algorithms for the decomposition of CFAs. We implement the exact algorithm in CPAchecker. We use the interface CFADecomposer promising a BlockTree when we input a CFA (Figure 11). In our case, a BlockTree knows the root BlockNode and returns all available blocks if requested. A BlockNode represents a subgraph of the CFA as defined in Definition 2 (*code block*). For implementation purposes, every CodeBlock knows its predecessors (Definition 3) and successors (Definition 4).

[1] `https://cpachecker.sosy-lab.org/`

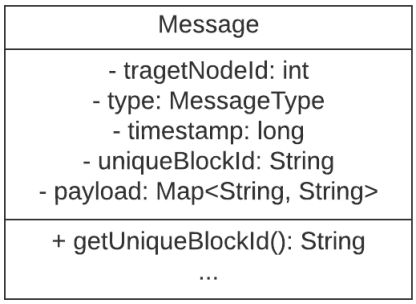


Figure 12: UML diagram of a message

5.1.2 Actor Model

Messages

Figure 12 shows the UML diagram of messages. They have the exact same attributes as discussed in Definition 8. Additionally, we store the time point of creation in the variable timestamp. An instance of the class Message is immutable and all attributes can only be accessed via getters. To keep the diagram simple, we omit the getter methods for the other variables although they are present. In addition to the messages, we implement a *message serializer* and a *message deserializer*. The *serializer* converts every message to a String using the JACKSON JSON library[2]. The *deserializer* implements the inverse function, also with the help of JACKSON. We need to be able to (de)serialize messages to send them over the network. Since messages use only primitive types like int, long and string, this is not difficult to do with JACKSON. The real work is done by the *deserialize operator* afterwards. The attribute payload imitates a dictionary that converts easily to JSON as demanded in Definition 8 (message). The attribute type is a member of the **enum** MessageType (one of BlockPostcondition, ErrorCondition, ErrorConditionUnreachable, Result, Error) and indicates the type of the message. The analysis processes messages depending on their types.

Connection

To make the communication with messages work, we need some sort of connection for the communication between our actors. We introduce the interface Connection for this purpose. A connection reads and writes messages, i.e., broadcasts it to all other connections. Furthermore, it knows the number of pending messages (size()). For convenience, we implement a default

[2] `https://github.com/FasterXML/jackson`

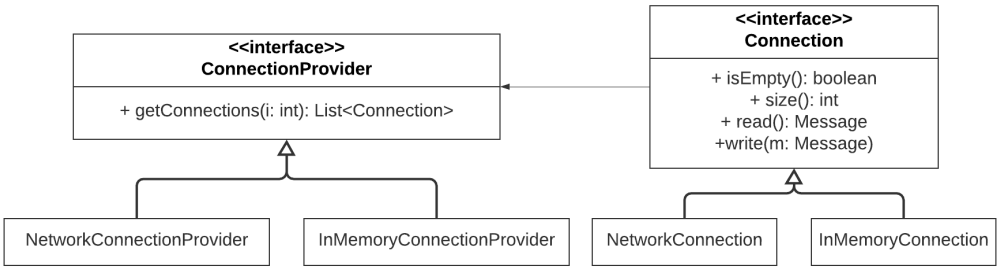


Figure 13: UML diagram of the connection types

routine for isEmpty(), namely isEmpty(){**return** size() == 0;}. Since every connection needs to know all other connections, we implement a *connection provider*. *Connection providers* return a specified number of connections where the write method of each connection, broadcasts the message to all of the other connections including itself. Since every actor is treated equally, the creation of connections must not know anything about the environment. Consequently, its sufficient to supplement the connection provider solely with the number of required connections. Figure 13 shows the UML diagram of the *connections* and the *connection providers*. There are two different types of connections: the *network connection* and the *in-memory connection*. For every existing worker, the *network connection* spawns a thread running a socket. The socket is responsible for receiving and storing messages in a blocking queue to which our actor has access via read(). Additionally, a *network connection* maintains n clients, each of which connects to one worker. Calling write lets every client send the deserialized message to the sockets. To receive messages at any time, the sockets run parallel. We use objects of the class ServerSocket as receiver and objects of the class Socket as sender. Both classes are part of the Java's standard `net` library. We encode messages to a byte-arrays as there already exist implementations for sending byte-arrays over sockets. After the *server socket* decodes the byte-array, the messages are put to the blocking queue. Whenever an actor calls nextMessage() we call take(), a method of Java's blocking queues. In case, the actor has no messages to process, take() blocks the actor and only releases it when a new message is present. Otherwise, the actor directly gets one of the pending messages. To reduce the size of the message we use Java's built-in GZIP in- and outputstreams for compression. Luckily, they operate on byte-arrays, too.

The *in-memory connection* creates one blocking queue for every actor. Afterwards, it collects every blocking queue in a list and copies the list to every connection as outgoing connections. By default, Java passes objects by

reference, i.e., we copy the reference and point to the same queue resulting in n queues for n workers. The method write adds a reference of the current message to every created queue. A major advantage of the *in-memory connection* is that we only need to allocate the space for one message once. Contrary, the *network connection* parses the byte-array and creates a new instance of the class Message to create a copy for each received message. The method size() returns the number of pending messages in the blocking queue.

We also provide an implementation using the class SocketChannel of Java's non-blocking IO package (`nio`[3]). However, message buffers of the non-blocking IO package do not guarantee to receive complete message at once. In our case, messages with over 200,000 bytes cannot be restored completely. To get rid of this problem, we have to manually manage the complete process of sending and receiving. As long as no reliable solution is available, we use *server sockets* instead.

Message Observer

The main thread maintains a list of message observers. In an endless loop, all observers match newly received messages against a pattern and collect information. The process method of an observer returns *true* if the endless loop should be exited, otherwise *false*. Whenever we break out of the loop, we print the collected results of all observers to the user. One observer, for example, tracks messages with type $\tau = \textsc{Error}$ and exits as soon as one message matches that pattern. Afterwards, all observers print their status.

Worker

Our workers inherit from the abstract class AbstractWorker. The class provides default implementations for all but the processMessage method. We briefly explain the implementation of the methods. Workers process messages one by one. The method nextMessage provides the workers with the next enqueued message. By default, it simply calls the (blocking) method read of connection. The connection can be changed with the setter because all workers have to be created without a connection before we can ask the *connection provider* for all connections. New workers are created with the help of the ComponentBuilder ensuring workers with a valid connection. For this reason, workers cannot be created manually and their classes are package-private. To implement a new worker, one has to extend the ComponentsBuilder with a corresponding method for creating this worker. The builder then takes care of the connections without further adaptions. The method broadcast loops through every

[3] `https://openjdk.java.net/projects/nio/`

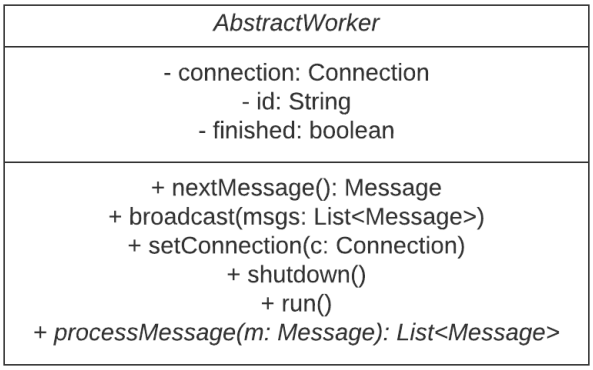


Figure 14: UML diagram of workers

message in the list and calls connection.write(m) with the message as parameter. The AbstractWorker realizes the interface Runnable (Java standard) and implements its method run. In the run method we implement the parallel tasks. In our case, we run Algorithm 5 for every worker by default. While we are not finished, we wait for the next message, then process it and afterwards broadcast our responses. Workers shutdown, whenever they receive a message of type Error or Result (c.f., Table 4). To shutdown a worker, we have to close the connection and set the flag finished to true. The method processMessage generally is implemented as a switch-case statement over the type of the input message m. Depending on the type, we start a backward or forward analysis or shutdown the worker. *Analysis worker* create completely independent forward and backward CPAs. In our case, that always is a composite CPA running the LocationCPA, BlockCPA, PredicateCPA and CallstackCPA. CPAchecker runs additional CPAs capable of identifying target locations. Target locations are either error locations or the initial or final locations of *code blocks*. Usually, CPAchecker abstracts predicate abstract states at target locations. Here, we prevent the abstraction of predicate abstract states as the abstraction equals *true* for most small blocks and we loose valuable information. Imagine a block B_1 with the code code x = 0; error;. The backward analysis reaches the initial location of the block and abstracts it to *true*, losing the information that x is required to equal 0 to reach the error.

5.2 Distributed CPAs

The AbstractDistributedCPA (Figure 15) is an abstract class that describes the behavior a distributed analysis should have. Distributed CPAs are initial-

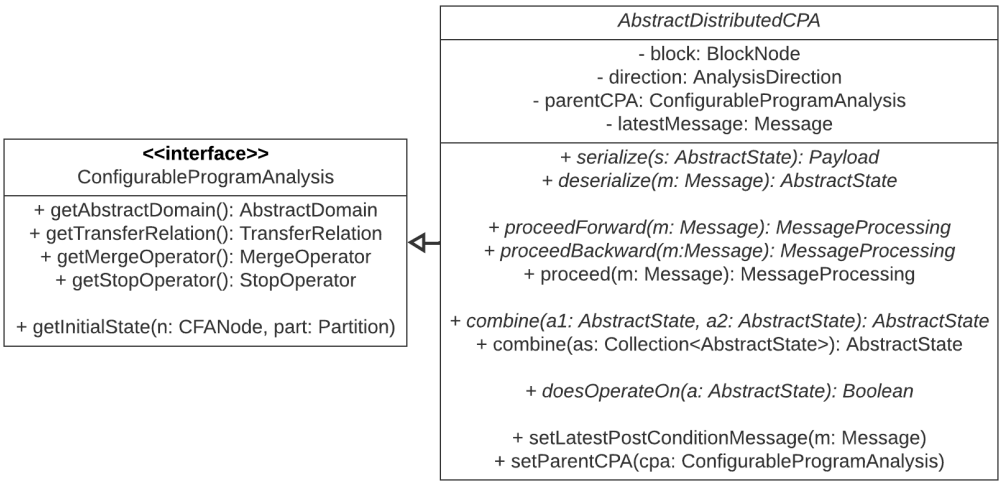


Figure 15: UML diagram of abstract DCPAs

ized with a block (*code block*) and the direction (forward or backward). The underlying CPA $\mathbb{C}$ is stored in parentCPA and can be set over the setter at the bottom of the diagram. We cannot create the CPA within the constructor of distributed analysis since CPAchecker builds the underlying CPAs in advance using reflection. The abstract distributed CPA implements the interface ConfigurableProgramAnalysis of CPAchecker. The interface defines a CPA. By composition, one member of the AbstractDistributedCPA is the parentCPA and thus it can forward its implementations of the methods of the interface directly. In other words, the methods of the interface ConfigurableProgramAnalysis for every AbstractDistributedCPA are implemented by calling the exact same method on the *parent CPA*. The method getInitialState returns the $\top$-element of the CPA. Additionally, every worker stores the result of the latest forward analysis in latestMessage needed for proceedBackward to check whether the current error condition can be reached through this block. Now, we describe the implementation of the characteristic operations of DCPAs. The method serialize takes an abstract state, realizing the interface AbstractState and translates it to a Payload representing a hash map. Generally speaking, serialize returns {parentCPA.getClass().getName(), toDeseralizableString(s)} where toDeseralizableString converts an abstract state to a string and the key equals an unique string identifying the underlying CPA. In our formal definition of distributed CPAs serialize produces messages instead. We could easily refactor the AbstractDistributedCPA to do this as well but that causes us to dissemble and re-assemble lots of messages in a composite distributed CPA. We would have to extract every immutable payload from every message

of one of the composite DCPAs and put it together to one payload and then again into a new message. The method deserialize takes the payload of a message and looks for the value mapped to the key parentCPA.getClass().getName() to deserialize it. In case, no such value is present, deserialize returns $\top$. In the implementation, we differ between proceedForward and proceedBackward to increase the readability of the code. As Java does not allow multiple return types, we combine the Boolean value and the collection of messages, promised by the *proceed operator*, in the class MessageProcessing. The class forwards a collection. Thus, we can iterate over instances of that class and we can ask whether we should stop the analysis. For this, instances of MessageProcessing store a Boolean value end, either set to *true* or *false*. The default implementation of proceed equals **return** direction == FORWARD ? proceedForward(m): proceedBackward(m);. The combine operator over sets exactly implements Figure 3. Child classes only have to implement the combine operator for two abstract states. Additionally, every abstract CPA knows, on which types of abstract states it operates. We can call the method doesOperateOn with the class of an abstract state to check this. We will see, why this is necessary in Section 5.2.2.

5.2.1 Distributed Predicate CPA

We continue with the description of the implementation of the distributed predicate CPA. The FormulaManagerView of CPAchecker already provides a parser of Boolean formulas (path formulas without SSA map) keeping the (de)serialization simple. The serialization reads the current path formula of the abstract state, uninstantiates it with u, and dumps the formula to a StringBuilder. We then put the key PredicateCPA.class.getName() together with the value of the string builder in a new Payload (representing a hash map) and return it. The deserialization reads the string mapping to the same key and parses it with the help of the FormulaManagerView and creates the SSA map by mapping every variable that does not contain a "." to the index 0. The method proceed works exactly as defined in Section 4.2 but it is composed of two functions each handling one case (forward/backward analysis). Case 3 (ignored message types) needs no implementation as we can just return (*false*, {}). The combination of two predicate abstract states works by making a disjunction of the path formulas of both inputted abstract states. The distributed predicate CPA operates on predicate abstract states and all child classes. For this, we just check whether the inputted abstract state is assignable to a predicate abstract state using reflection.

5.2.2 Distributed Composite CPA

A composite CPA simultaneously runs different CPAs on the same CFA. Whenever we want to use a DCPA in a distributed composite CPA, we have to register the new DCPA first. We implement it in a lazy way. The hash map lookup maps the class of an existing CPA to the class of the corresponding DCPA. In case, a CPA is part of the parent composite CPA and happens to be a key in lookup, we create the distributed CPA with the reflection API of Java. In other words, we only create instances of distributed CPAs when we really need them. If no distributed CPA is implemented for a given CPA, the information will not be serialized. We implement the serialization of a composite abstract state by serializing each contained abstract state with the help of known DCPAs operating on this abstract state. That is the reason why we need the function doesOperateOn. Information of unimplemented DCPAs is lost and cannot be restored on other workers. The deserialization of a composite state works analogously. Since we at least know the potential parent CPA of unimplemented DCPAs we can simply use the respective $\top$-element by calling getInitialState (...) on it.

A distributed composite CPA proceeds the calculation if all contained DCPAs proceed. Unimplemented DCPAs proceed with the empty set of messages by default. One DCPA suffices to stop the calculation if it cannot proceed. However, we respond with the union of all messages returned by the *proceed operator* of every single DCPA.

The combination of two composite abstract states works by combining the abstract states of all contained DCPAs with their *combine operator*. In case, no such DCPA exists, we return the respective $\top$-element as in the deserialization.

The distributed composite CPA exclusively operates on abstract states that are an instance of the class CompositeState.

5.2.3 Distributed Callstack CPA

Instead of inlining every function call in the CFA, CPAchecker creates an own CFA for each function and maps a *function call edge* from the caller-statement to the function and a *function return edge* form the function to the caller-statement. The *callstack CPA* tracks from which location a function is called and only transitions a *function return edge* if it maps back to the correct caller-statement. To mimic this behavior, we have to implement a distributed *callstack CPA*. We exclusively use the *callstack CPA* for the backward analysis because it is the most precise analysis running. Assume, a block represents a complete function. Then, the precondition for the forward

analysis equals *true* as long as an existent function call has not been made yet. If all function calls have been made by the forward analysis, its precondition equals the disjunction of the inputs of all function calls which does not lead to an under-approximation and thus is valid. The abstract states of the callstack CPA are stacks, storing all function calls and the location from which the function call originated. Hence, we serialize the abstract states by separating all tuples of the location and the function name with commas, e.g., main 16, function1 20. The given state tells us that we called the function function1 on location 20 after we entered the main-function on location 16.

5.2.4 Block CPA

Instead of creating a new CFA for every block, we simply run the BlockCPA in a composite CPA with the predicate CPA on the original CFA. The BlockCPA works identical to the location CPA with the restriction that it will not allow a transfer if we would leave our block. There is one minor adaption: if a block starts and ends with the exact same location, i.e., the block is circular, we only transfer once from the loop head. Hence, every iteration issues one message. This is necessary to 1) prevent an infinite analysis and 2) to keep other actors up-to-date because we gain information after every iteration that we do not want to hold back. The BlockCPA realizes the interface ConfigurableProgramAnalysis of CPAchecker and thus implements the two operators and the transfer relation discussed in Definition 9, primarily reusing the operators of the location CPA $\mathbb{L}$.

5.2.5 Determining the Verification Result

As mentioned above, workers implement the interface Runnable and thus can be run within a thread. Our main thread spawns all threads for the workers and connects via an extra connection to the workers to listen for result messages. As soon as a worker broadcasts a verification result, the main thread prints it to the user. In case of active *fault localization* the main thread additionally collects and eventually prints all error-prone edges. CPAchecker prints True as verification result if no unfinished abstract states are contained in the reached set and none of the abstract states reaches an error location. Currently, we do not collect the great amount of computed abstract states across the workers and put them together in the reached set of the main thread. We simply empty the reached set of the main thread causing CPAchecker to print True. For violations, we empty the reached set and manually put a dummy target state in the reached set. Otherwise, we do not modify the reached set, leaving the - in the main thread unprocessed

- initial state in it. Since it is unprocessed, CPAchecker prints Unknown as verification result.

5.3 Distributed Fault Localization

To execute the distributed fault localization, we extend the *analysis worker* by one simple addition. Before we broadcast a new *error condition*, we execute the existing MaxSat algorithm in CPAchecker [17] on the block. The pre-condition equals the model of the error-condition in conjunction with the path formula representing the current block as described in Section 4.3. Whenever the first backward analysis transitions an assume edge, we create a new instance of the MaxSat algorithm and run it on that block. The post-condition equals the path formula of the assume edge closest to the block end. Then, we can run the fault localization. Afterwards, we store the post-condition and all error-prone edges in ρ and broadcast the message. Predecessor blocks use this information in ρ to search for the exact statement in the *error condition* and substitute it by its negation to ensure the desired behavior. Our main thread monitors all messages. To print the fault localization result to the user, we register the *fault localization message observer* that collects all found faults. At the end, it prints the error prone locations to the user.

5.4 Configurations

Our implementation is highly configurable. Table 12 lists all possible configurations and their default values. The *decomposition type* decides whether we use the LinearDecomposition (`BLOCK_OPERATOR`) or the GivenSizeDecomposition (`GIVEN_SIZE`). The desired number of blocks is only relevant if the decomposition type is set to `GIVEN_SIZE`. The decomposition can be disabled with `SINGLE_BLOCK`, resulting in only two workers: the *root worker* and one *analysis worker* for the whole CFA. The worker type decides what type of workers run on the *code blocks*. `DEFAULT, SMART` and `MONITORED` run the distributed CPA on a given program. The option `SMART` lets the workers discard unimportant messages immediately and it combines all received BlockPostcondition messages to one, i.e., it uses *fast analysis worker* (Section 4.1.6). The monitored worker should not be used in practice but comes in handy for debugging purposes. A global monitor watches all workers and lets only work a given number of workers at the same time. Other workers are blocked by the monitor. The worker type `FAULT_LOCALIZATION`

Table 12: All configurations

configuration	possible values	default
decompositionType	`SINGLE_BLOCK`, `BLOCK_OPERATOR`, `GIVEN_SIZE`	`BLOCK_OPERATOR`
workerType	`DEFAULT, SMART, FAULT_LOCALIZATION, MONITORED`	`DEFAULT`
connectionType	`IN_MEMORY, NETWORK`	`NETWORK`
desiredNumberOfBlocks	any integer number	10
maxWallTime	time span (e.g., 15min)	900 s
daemon	*true, false*	*true*
spawnUtilWorkers	*true, false*	*true*
flPreconditionAlwaysTrue	*true, false*	*false*
abstractAtTargetState	*true, false*	*false*
checkEveryErrorCondition	*true, false*	*true*
sendEveryErrorMessage	*true, false*	*false*

runs an analysis worker with additional fault localization. With the *connection type*, we either activate the communication over the network or we use the in-memory communication. Finally, users can set the desired number of blocks s for the GivenSizeDecomposition with the option *desiredNumberOfBlocks*.

The *timeout worker* and the *visualization worker* can be disabled by deactivating the option *spawnUtilWorkers*. The wall time limit for the *timeout worker* can be set via the option *maxWallTime*. As already mentioned before, our workers run in their own threads. Threads can be marked as daemons causing the threads to shut down whenever the main thread terminates. Although our implementation should not leave threads running, we provide this option to make sure that all resources are freed and all threads are shut down.

The option *flPreconditionAlwaysTrue* lets *fault localization workers* skip the computation of a model to create the precondition. Every precondition is set to *true* initially. Calculating a model can also be expensive and in some cases it is not necessary. Generally, the precondition only prevents initial assignments to be part of the faulty statements.

The option *abstractAtTargetState* runs the user-defined abstraction when

```
-predicateAnalysis
-setprop analysis.algorithm.configurableComponents=true
-setprop cpa.predicate.blk.useLoopStructure=true
-setprop cpa.predicate.blk.alwaysAtJoin=true
-setprop cpa.predicate.blk.alwaysAtBranch=true
-setprop cpa.predicate.blk.alwaysAtProgramExit=true
-setprop limits.time.cpu=1h
-setprop components.decompositionType=GIVEN_SIZE
-setprop components.connectionType=IN_MEMORY
-setprop components.workerType=DEFAULT
<path to program>
```

Figure 16: Possible configuration to run the distributed predicate analysis

reaching an error location or the final or initial location of a block. In the future, this is meant to decrease the size of the formulas, as only to the error important parts of the formula are kept. Currently, the solver of CPAchecker has problems dealing with the abstraction type Elimination such that we currently use the *boolean abstraction*. However, the Boolean abstraction always returns *true* as abstraction formula and we loose all information. As soon as *elimination* is supported, we can activate this option by default.

The option *checkEveryErrorCondition* forces the proceed operators to check every error condition for satisfiability. In the following chapter, we will see that this is not always desirable as the SAT-checks are very time-consuming. Setting this option to *false* allows the *proceed operator* to skip the SAT-checks for workers with more than 3 predecessors.

Disabling the option *sendEveryErrorMessage* prevents workers from sending *error condition* messages multiple times. In case, the input program contains loops, the options will automatically be enabled. For loop-free programs, broadcasting the exact same messages causes the exact same computation steps and thus can be saved. A more detailed explanation follows in Section 6.3.3. Figure 16 shows a possible configuration.

Table 13: Message types and their desired precedence

message type	prio
FoundResult	1
Error	2
ErrorCondition	3
ErrorConditionUnreachable	4
BlockPostCondition	5

5.5 Message Prioritization

To speed up the distributed analyses, we implement a prioritization of messages, shown in Table 13. The connection has to sort pending messages ascending by their priority. Java provides a PriorityBlockingQueue that inserts all new messages sorted. Although, according to the official Java Doc[4], the runtime complexity of inserting elements in a priority queue is $O(\log n)$, we experience a performance boost. Whenever an error occurred or a result has already been found, we should print the results instantly. We do not have to wait until the workers have processed a potentially large number of other messages first since a *found result* message is definite and guaranteed to be correct, every further work would be unnecessary. The same holds for errors and exceptions. The reason for prioritizing *error condition* messages over *block post condition* messages is the rapid propagation of errors to the root worker for identifying possible violations early.

5.6 Visualization

We implement a visualization of the messages for debugging purposes. The *visualization worker* logs every message in JSON format to a JSON file. A python script transforms the contained information into a HTML table and a block graph (DOT format). The columns represent the workers and the rows the passed time. The contents of the table are the messages sent by a worker at a certain time point. Messages of type BlockPostcondition are colored in yellow. *Error conditions* are colored red and results are colored green. The example in Figure 17 shows the answer of worker W_1 (operating on B_1) to the initial message of our root worker after 1 millisecond. The

[4] `https://cr.openjdk.java.net/~iris/se/12/latestSpec/api/java.base/java/util/PriorityQueue.html`

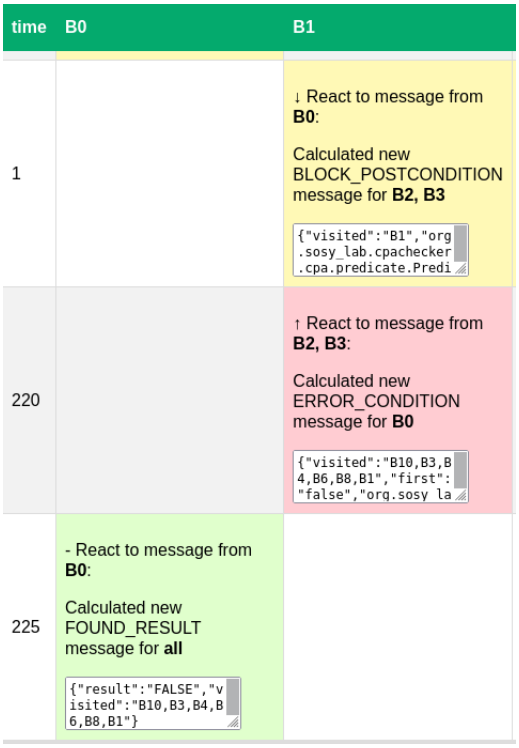


Figure 17: Second, penultimate and last row of the visualized log

penultimate row shows that the *error condition* holds at the end of block B_1 and thus W_1 started a backward analysis. Finally, the root worker proves the *error condition* feasible in the last row, resulting in the verification result FALSE, i.e., the program is unsafe. The messages are sorted descending by the time of creation. Simultaneously created message are in the same row of the table.

6 Evaluation

6.1 Setup

We benchmark our implementation on a subset of the ReachSafety tasks from SV-COMP 2022 [3]. We use two setups for our experiments:

Setup 1 Intel Xeon E3-1230 v5 processor with 8 CPU cores at 3.40 GHz each and 32 GB RAM.

Setup 2 Intel Core i7-10700 processor with 16 CPU cores at 2.90 GHz each and 67 GB RAM.

In total we use 6671 tasks from the following sets:

- ReachSafety-BitVectors,
- ReachSafety-ControlFlow,
- ReachSafety-ECA,
- ReachSafety-Heap,
- ReachSafety-Loops,
- ReachSafety-ProductLines,
- ReachSafety-Sequentialized,
- ReachSafety-XCSP,
- ReachSafety-Combinations and
- SoftwareSystems

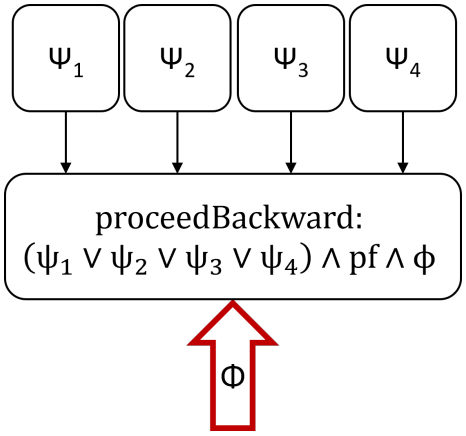


Figure 18: A worker with 4 predecessors has to check the satisfiability of 4 preconditions together with one post-condition. This takes longer than just running a backwards analysis without satisfiability check and forwarding the satisfiability check to the predecessors. The checks are then parallelized which saves time. Additionally, the number of atoms in a disjunction decreases. The formula *pf* denotes the disjunction for all possible paths from the initial location to the final location of a block.

All experiments are run on revision f404760a of CPAchecker[1]. The artifacts are available here[2]. CPAchecker provides a script for benchmarking a given configuration with BenchExec [21].

To improve the performance of the distributed analysis, we activate three optimizations by default for all following benchmarks. First, we slightly change the behavior of the *proceed operator* for the distributed predicate analysis. Instead of returning (*true*, {}) we return (*false*, {}) whenever $\mathbb{F}$ is incomplete (Case 1.1.2 in Section 4.2) since we would send the exact same message as in the initial analysis. Hence, we do not gain new knowledge and we can omit the message to reduce the traffic. Second, we apply the message prioritization described in Section 5.5 and we skip SAT-checks if $|\mathbb{F}| > 3$. If the cardinality of $\mathbb{F}$ is large, the precondition of a worker contains many disjunctions. SMT-solvers have problems dealing with disjunctions as they have to check every possibility. For our distributed approach, it is more efficient to just issue a backward analysis for workers with more than three predecessors and let the predecessors execute the checks parallely instead. Figure 18 illustrates the performance boost. To justify the usefulness of the three optimizations, we run our DCPA on *linear code blocks* with and

[1] `https://gitlab.com/sosy-lab/software/cpachecker/-/commit/f404760a`
[2] `https://doi.org/10.5281/zenodo.6224978`

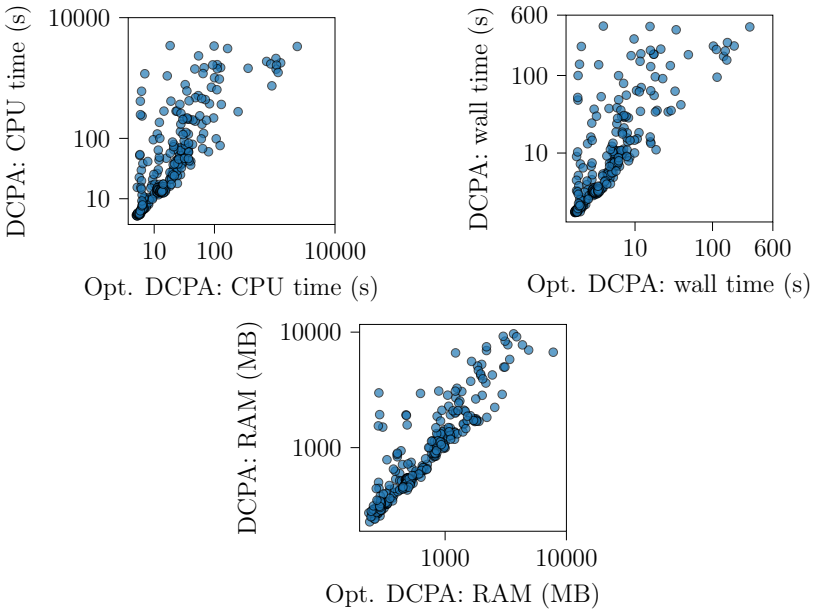


Figure 19: Comparison of DCPA_D with optimizations (x-axis) and without optimization (y-axis).

without them. We use scatter plots to visualize the results. Scatter plots plot all values of the same measurement produced by two different candidates for the same task as x and y value, respectively. Every data point represents a verification task that both analyses verified correctly. We compare the performance of an analysis with regard to the wall time, the CPU time and the memory usage. Figure 19 displays three scatter plots with the optimized DCPA on the x-axis and the DCPA without optimization on the y-axis. The data points are above the diagonals in all plots, proving the positive effect of the optimizations. For many of the tasks the optimized DCPA solves in under 100 seconds, the unoptimized DCPA takes far more time. The same is valid for the CPU time. The memory usage improves but the effect is not as strong as for the CPU or wall time. Moreover, the unoptimized DCPA produces more timeouts and out of memory exceptions.

6.2 Experimental Results

We compare our approach with different configurations to the already existing predicate analysis in CPAchecker. Currently, our approach does not support pointer-aliasing. Therefore, we deactivate it for the existing predicate analysis, too. In this section, we first present an overview of the results

Table 14: All benchmark configurations

	predicate	DCPA-	DCPA$_D$	DCPA↓	DCPA↓+
distributed?	✗	✓	✓	✓	✓
optimized?	-	✗	✓	✓	✓
decomposition	-	linear	linear	given size	given size
worker type	-	default	default	default	fast

Table 15: Occurrence of different statuses.

status	predicate	DCPA$_D$	DCPA↓	DCPA↓+
out of memory	39	1417	2108	2118
timeout	1920	2907	2001	2003
error	1539	1586	1935	1929
TRUE	2086	504	382	375
FALSE	1087	257	245	246
$\sum$	6671	6671	6671	6671

of each configuration and afterwards, we continue with an in-depth analysis of the advantages and disadvantages of the configurations. All tasks have a wall time limit of 10 minutes. Until further notice, all benchmarks are executed on **Setup 1** with the *in-memory connection*. We do not spawn utility workers (*visualization worker* and *timeout worker*) to decrease the number of required connections. In addition, the main purpose of the utility workers is to assist with debugging. Hence, they do not contribute to the verification process.

6.2.1 Overview

Table 15 shows the results of the benchmarks for each configuration. The column *predicate* shows the results of the existing predicate analysis and DCPA$_D$ shows the results of the distributed predicate analysis without further improvements. DCPA↓ indicates the results of the distributed predicate analysis but we use the GIVENSIZEDECOMPOSITION instead of the LINEARDECOMPOSITION. The column DCPA↓+ contains the results of the distributed predicate analysis with the GIVENSIZEDECOMPOSITION and *fast workers* instead of normal *analysis workers*. The different configurations are shown in Table 14. We will now briefly cover the main takeaways from Table 15. The first row shows the number of tasks where the verification does

Table 16: Soundness of DCPAs

result	predicate	DCPA$_D$	DCPA↓	DCPA↓+
correct	2993	675	548	544
incorrect	180	86	79	77

not finish due to an exceeded memory limit. We observe that the distributed approach uses more memory than the predicate analysis. The problem arises because of the messages and the parallel analysis. The distributed approach stores all messages for all workers until they are processed, whereas the predicate analysis never stores unimportant information. The second row shows the number of timeouts. The distributed approach takes more time. The reason for this is the high number of SAT checks. More on that topic follows later. The third row shows the number of unexpected exceptions causing the analysis to stop. The gap of 400 more exceptions for DCPA↓ and DCPA↓+ compared to DCPA$_D$ can be tracked back to two reasons. First, the default backward predicate analysis is not as precise as the forward predicate analysis, especially with the handling of pointers and arrays. Second, the backward CallstackCpa is more error-prone and throws more exceptions. Rows 4 and 5 show the number of solved tasks with the respective verification result of True and False. For completeness, the sixth row shows the sum of tasks to ensure that we verified the same number of tasks for all configurations.

6.2.2 Soundness of DCPAs

The benchmark set provides us with the expected verification result for all tasks in the set. Table 16 shows the number of correct and incorrect verification results, i.e., tasks where the verification wrongfully proves a program (in)correct. We are especially interested in the incorrect results. First, we check, whether the incorrect results of the distributed approach match the incorrect results of the existing predicate analysis. DCPA↓ and DCPA↓+ do only report wrong results if the predicate analysis does, too. DCPA$_D$, however, contains 8 tasks where the verification result diverges. 7 out of the 8 tasks occur in the same class of verification tasks, namely, *productlines/minepump*. The existing predicate analysis correctly solves these tasks, whereas the DCPA reports wrong proofs. The DCPAs with the GivenSizeDecomposition throw an exception for these tasks claiming that the program contains the unsupported feature *recursion*. Presumably, the linearity of the blocks prevents the recognition of the problem for the backward analysis. We assume that the problem originates form the backward anal-

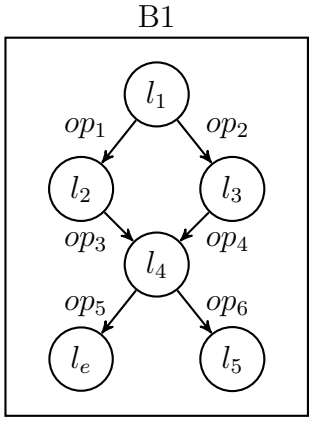
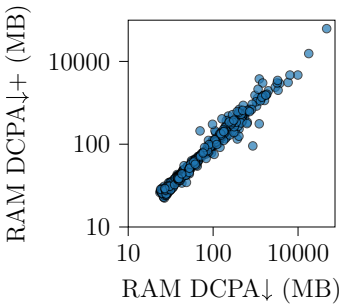


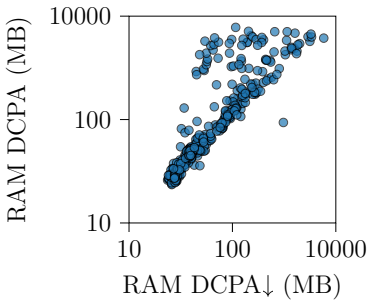
Figure 20: The backward analysis returns a disjunctive path formula for error locations in merged blocks structured like this.

ysis from which the existing predicate analysis does not make use of. The remaining tasks uses the CPAchecker internal *pointer target set* which is not yet part of the messages. The preparation of the *pointer target set* for the transport over messages is difficult as the pointer target set consists of various complex data structures where no easy way for (de)serialization exists until now. However, the results reinforce our belief that the implementation is correct and that the approach works. Currently, the approach is also limited by the hardware resources. In addition, the following sections explain possible improvements for the distributed approach.

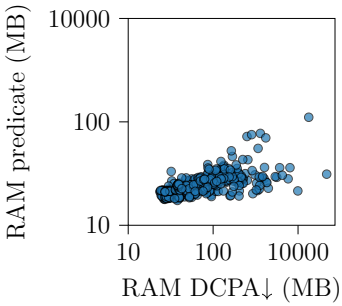
DCPA_D with linear decomposition finds over 200 proofs more than workers operating on merged blocks. We identify two reasons for this behavior: first, the backward callstack CPA recognizes recursions on merged blocks although there are none. The linear decomposition prevents these false alarms since the *proceed operator* prohibits many of the backward analyses causing the CallstackCPA to throw the exceptions. Secondly, the finer division into blocks rejects infeasible paths earlier, saving computation time. Figure 20 shows a merged block with an error location at the block exit. The backward analysis creates the path formula $op_5 \wedge ((op_3 \wedge op_1) \vee (op_4 \wedge op_2))$, starting form the error location l_e, because the *merge operator* of the predicate CPA creates a disjunction of the two possible paths at the block entry l_1. For every backward analysis, we have to check the disjunction of the precondition with the disjunction of the *error condition* for satisfiability. Linear blocks causes workers to analyze every possible *error condition* separately, whereas path formulas at the initial location of a merged block might represent a set of possible error paths. Denying a set of *error conditions* usually



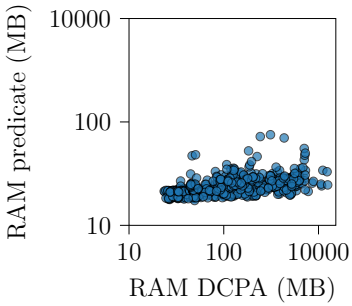
(a) Comparison of RAM usage between DCPA↓ on the x-axis and DCPA↓+ on the y-axis.



(b) Comparison of RAM usage between DCPA↓ on the x-axis and DCPA on the y-axis.



(c) Comparison of RAM usage between DCPA↓ on the x-axis and the predicate analysis on the y-axis.



(d) Comparison of RAM usage between DCPA and the predicate analysis on the y-axis.

Figure 21: Comparison of the memory usage

requires more knowledge. Hence, the *error condition* has to be propagated further. Furthermore, the resulting SAT-checks are time-consuming.

In the following three sections, we compare the different configurations (Table 14) with respect to the memory usage, the needed wall time and the needed CPU time.

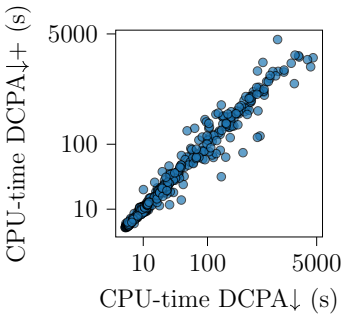
6.2.3 Comparison of Memory Usage

We begin with the comparison of the RAM usage and exclusively consider the tasks where the compared configurations of the analyses are correct. First, we compare the existing predicate analysis to DCPA_D. The corresponding scatter plot is shown in Figure 21d. The distributed predicate CPA is shown on the x-axis. We plot the memory usage on a logarithmic scale. The unit is given in mega byte (MB). The memory usage of the distributed approach is

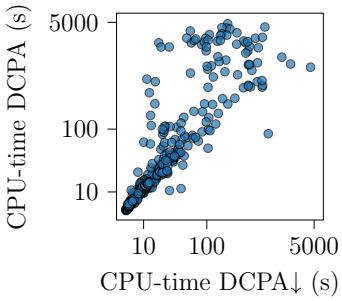
higher. The reason for this is the message exchange since messages rapidly grow up to over 200,000 characters. In case, many workers are running, the number of messages and thus the required amount of memory increases. Additionally, many messages contain information about infeasible paths but are only discarded at a very late stage because the workers do not guarantee a certain order of processing messages. We introduce the GivenSizeDecomposition to reduce the number of workers and thus the number of messages. The plots shown in Figures 21b and 21c support our assumption. Figure 21c compares DCPA↓ to the predicate analysis. We observe a left-shift of the data points indicating that the distributed approach with fewer workers needs less memory. The direct comparison of DCPA and DCPA↓ shows the same effect, the data points are in above the diagonal of the plot, meaning that the DCPA usually needs more RAM. Lastly, we compare the memory usage of DCPA↓ to DCPA↓+. The data points in Figure 21a are basically aligned in a straight line. Hence, the memory usage is equal. This is expected because a *fast worker* processes the exact same messages but triggers less analyses. The noise in the plot can be explained with the nondeterministic processing of messages.

6.2.4 Comparison of CPU Time

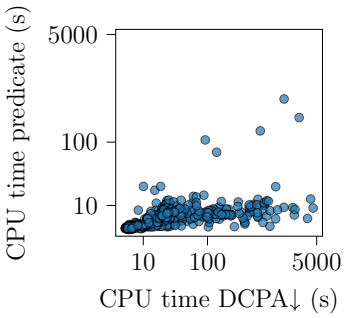
Next, we compare the CPU time. We expect the predicate analysis to perform better as no unnecessary work is done. Our workers check every message for satisfiability and therefore consume more CPU time. Additionally, many workers run parallel. The scatter plots in Figure 22 prove our assumption correct. DCPAs take significantly more CPU time, shown by the accumulation of the data points on the bottom right, beneath the diagonals of Figures 21c and 22c. We implement the GivenSizeDecomposition since we want to reduce the number of messages and subsequently speed up the analysis. Figure 22b indicates a slight improvement of the CPU time consumption over the distributed analysis with the *linear decomposition*. The reduction of the *code blocks* and the accompanying reduction of *analysis workers* helps finding the results faster. Additionally, the number of timeouts goes back, too. Table 15 provides evidence that there are 900 fewer timeouts when reducing the number of workers compared to the distributed analysis using the LinearDecomposition. Unfortunately, most tasks result in other errors that are not detected by the linear decomposition. We already described this phenomenon earlier. Figure 22a compares the CPU time of DCPA↓ with DCPA↓+. Their CPU times are also very similar because the forward analyses do not consume as much resources as the SAT-checks.



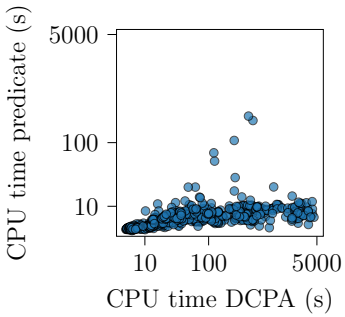
(a) Comparison of the CPU time between DCPA↓ on the x-axis and DCPA↓+ on the y-axis



(b) Comparison of the CPU time between DCPA↓ on the x-axis and DCPA on the y-axis



(c) Comparison of the CPU time between DCPA↓ on the x-axis and the predicate analysis on the y-axis

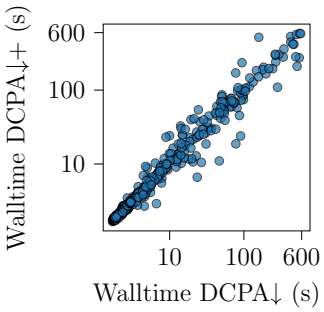


(d) Comparison of the CPU time between DCPA and the predicate analysis on the y-axis

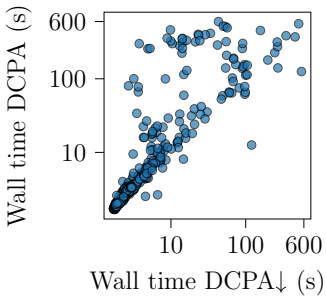
Figure 22: Comparison of the CPU time

6.2.5 Comparison of Wall Time

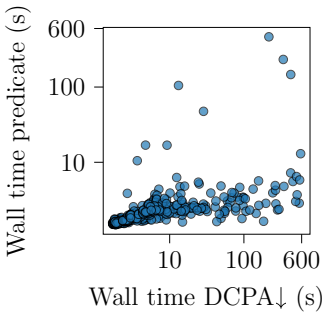
Finally, we compare the wall time of each configuration. Currently, the predicate analysis outperforms the distributed approaches. Figures 23c and 23d plot the wall time of the distributed approach (with and without GivenSizeDecomposition) and the time for the predicate analysis. Most of the points are beneath the diagonal. In other words, the predicate analysis is faster. The reason for this is the sequential processing of the messages. Every *error condition* needs time-expensive SAT-checks. Messages reach sizes of over 200,000 bytes. The biggest part of every message is the string representation of the formula that has to be parsed again. In case the precondition of a worker was never updated, the backward analysis tends to cause unnecessary work as we traverse a potential infeasible path. To sup-



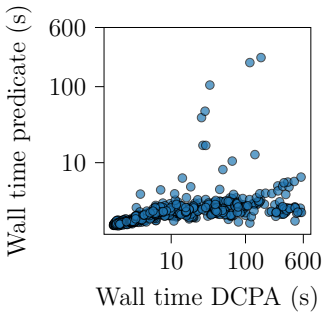
(a) Comparison of the wall time between DCPA↓ on the x-axis and DCPA↓+ on the y-axis



(b) Comparison of the wall time between DCPA↓ on the x-axis and DCPA on the y-axis



(c) Comparison of the wall time between DCPA↓ on the x-axis and the predicate analysis on the y-axis



(d) Comparison of the wall time between DCPA and the predicate analysis on the y-axis

Figure 23: Comparison of the wall time

Table 17: Average wall time for calling each operator once.

operator	DCPA	DCPA↓	DCPA↓+
serialize	0.62 s	0.87 s	1.37 s
deserialize	0.74 s	1.09 s	1.58 s
proceed (f)	1.66 s	4.73 s	3.89 s
proceed (b)	5.00 s	3.89 s	6.93 s
combine	0.19 s	0.27 s	0.33 s

port our argumentation, we measure the wall time for each operator of the distributed CPAs. For every program under analysis, we track the average time of all workers for all four operators for every task with a verification result of TRUE or FALSE. The averages of these time spans are listed in

Table 18: Comparison of average wall time (s) to find a correct proof or counterexample for the predicate analysis and the DCPA

	predicate	DCPA	factor
correct proofs	3.34 s	43.58 s	13.06
correct counterexamples	2.92 s	38.94 s	13.33

Table 17. The *proceed operators* take up to 7 seconds per call on average. The average time for (de)serialization is approximately 1 second, implying that large messages are sent. To conclude, the current approach needs a reduction of SAT-checks in the *proceed operations* as, for example, described in Figure 18. Possible improvements leave out SAT-checks based on previous results or only perform the check sporadically. The comparison of DCPA↓ and DCPA↓+ (Figure 23a) leads to the conclusion that combining the abstract states in advance is irrelevant. The forward analyses on a block do not take long. Thus, saving forward analyses has no impact regarding the wall time.

The distributed approach takes longer than the predicate analysis. We are interested in the average time to find a correct proof and a correct counterexample between both approaches. Table 18 shows that the predicate analysis takes about 3 seconds to find a correct proof. The distributed approach takes 43 seconds. For correct counterexamples, the predicate analysis needs approximately 3 seconds as well, whereas the distributed approach needs 38 seconds more. In general, finding a correct result takes about 13 times longer. We run the verification for all correctly solved tasks again on **Setup 2** to examine the effect of more hardware resources. In theory, our approach benefits from more computation power and more memory. Figure 24 shows three scatter plots listing the results of DCPA$_D$s on **Setup 2** on the y-axis and the results of **Setup 1** on the x-axis. The data does not provide sufficient evidence that increasing the resource limits indeed speeds up the analysis. We find a possible explanation in other collected statistics: more computational power causes an increasing number of messages. Whereas, on **Setup 1** 2,940 messages are sent on average, the average number of sent messages increases by 25% to 3,660 using **Setup 2**. Since we only consider correctly solved tasks, we know that all but one message (the *result message*) are either of type ErrorCondition, ErrorConditionUnreachable or BlockPostCondition. For most of these messages, we have to perform a time-expensive SAT-check.

To conclude, the predicate analysis outperforms the distributed approaches because of the abstraction. Therefore, the next section compares the dis-

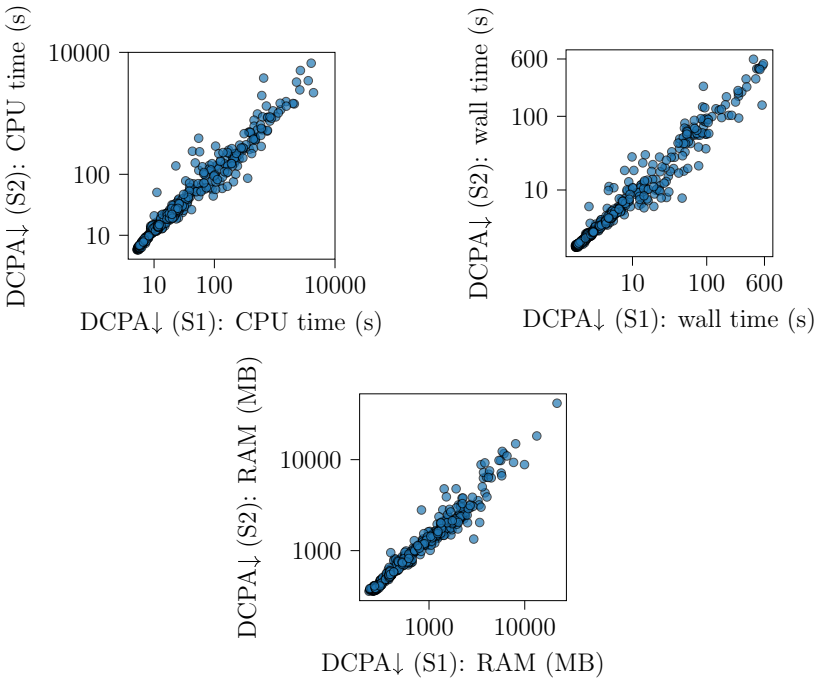


Figure 24: Comparison of DCPA_D running on **Setup 1** (x-axis) and **Setup 2** (y-axis).

tributed approach with *decompositionType* = `SINGLE_BLOCK` to DCPA_D to see the effect of the distribution.

6.2.6 Comparison to DCPAs with Single Worker

To show the impact of the distribution of the verification to many workers, we run DCPA_D on a single block containing the complete CFA instead of applying the *linear decomposition*. We refer to this configuration as DCPA(SB). Figure 25 shows the corresponding scatter plots with all tasks solved correctly by both configurations. DCPA(SB) is displayed on the y-axis. On the x-axis, we see the default DCPA (DCPA_D) with linear blocks. The plots show that the distribution speeds up the analysis significantly. The distribution on the x-axis finishes within 10 seconds, whereas the analysis on a single block takes up to 10 minutes for the same tasks. The number of timeouts also increases drastically. As shown in Table 15, DCPA_D times out 2907 times whereas DCPA(SB) causes 4776 timeouts. However, the number of out-of-memory errors decreases from 1,417 to 27 since the number of messages decreases drastically. In total, DCPA(SB) solves 183 tasks (36 incorrect) which is significantly less than the 761 tasks (86 incorrect) from DCPA_D.

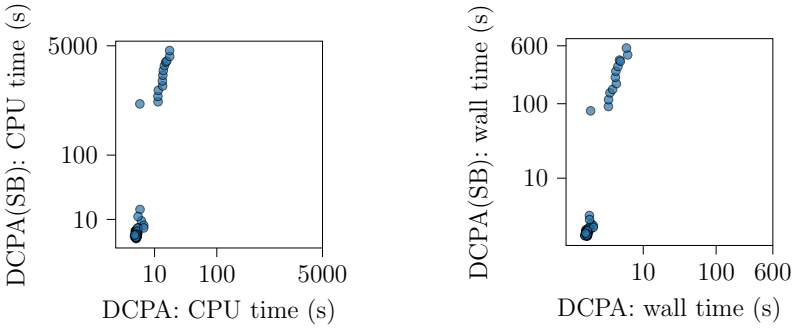


Figure 25: DCPA_D running on a single block (y-axis) versus DCPA_D running on multiple blocks (x-axis).

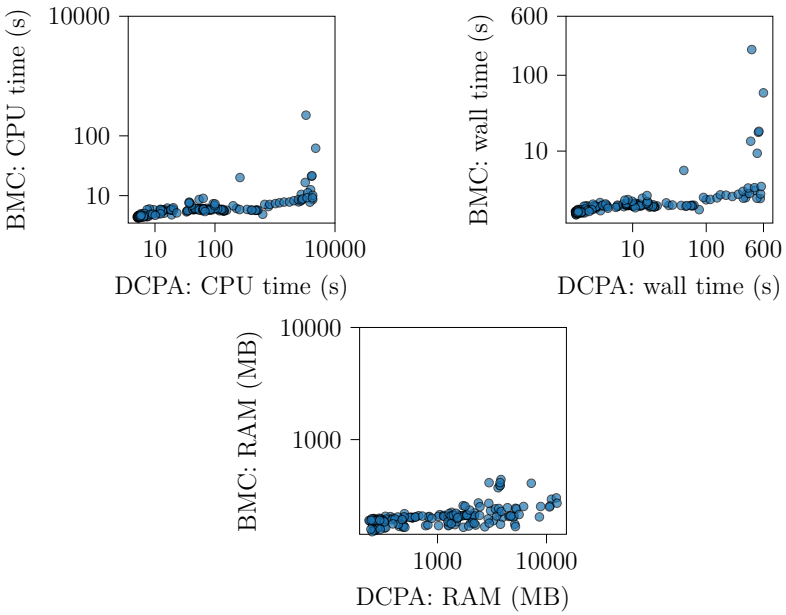


Figure 26: Comparison of DCPA_D (x-axis) with BMC (y-axis).

6.2.7 Comparison to BMC

Bounded model checking (BMC) [7, 10] executes the predicate analysis but unrolls loops only a given number of times. In case, the analysis cannot finish because the loops are not unrolled completely, BMC returns the verification result UNKNOWN. BMC solves approximately 800 tasks correctly but it is faster and uses less memory. Figure 26 shows three plots with our distributed approach (DCPA_D) on the x-axis and BMC on the y-axis. BMC solves most tasks in under 10 seconds, whereas DCPA_D takes up to 10 minutes. The

Table 19: The average time and memory usage for DCPAs with and without compression.

	without compression	with compression
avg. CPU time	273 s	269 s
avg. wall time	96 s	92 s
avg. memory usage	13,078 MB	12,939 MB

memory-usage ranges from 100 MB to 10 GB for the distributed approach. BMC always uses less than 1 GB of RAM.

6.2.8 Network Connection

In this section, we compare the DCPA↓ using the *in-memory connection* to the DCPA↓ using the *network connection* with and without message compression. We use the built-in GZIPInputStream and GZIPOutputStream to zip the messages. This reduces the size of the messages by approximately 30% on average. Compression does only help to a limited extend. While it reduces the size for the message transfer, it does not reduce the overall needed memory as we decompress messages to the original size on all ends again. Contrary to the *in-memory* connection, we cannot pass messages by reference. To send the messages over the sockets, we translate them into byte-arrays. The receiver of the byte-array transforms the byte-array back to an instance of the class Message. Hence, every receiver allocates memory for the exact same message. Since our implementation is only executed on one machine, we tend to exceed the memory limit faster. However, we can run our implementation on different machines to mitigate this problem. Currently, running experiments on different machines is not supported. We lack the implementation of a strategy, capable of distributing an equal amount of work to every connected machine. As soon as this is implemented, the existing *network connection* should work without further adaptions. Table 19 compares the *network connection* with and without compression. In many cases BenchExec did not interrupt the analysis after the given wall time limit of 600s (10 minutes) expired. Therefore, some data points contain large numbers for the CPU and the wall time. As a consequence, we only consider the tasks that have a wall time of 600s or less for this comparison. The average values of the CPU time, wall time, and RAM usage are nearly equal. The data does not show significant differences regarding the compression. Taking a look at Figure 27 supports this assumption as the data points are aligned in a straight line. The x-axis shows the results of the distribtued verification with com-

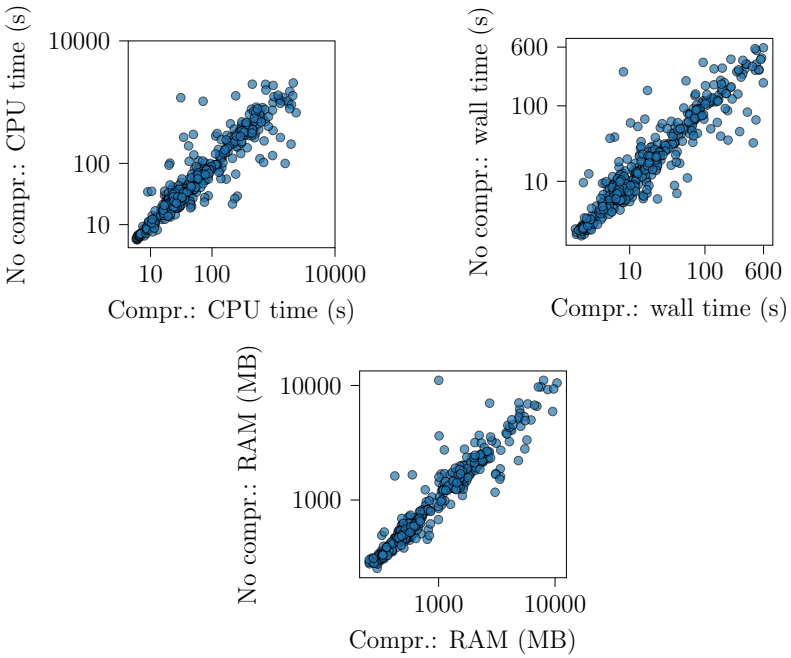


Figure 27: Comparison of the CPU time, wall time and memory usage of the DCPA↓ with and without compression when using the *network connection*.

pression and the y-axis shows the results without compression. Once again, the noise comes from the nondeterministic processing of messages. Finally, we examine the influence of the connection type on the performance of the verification. Figure 28 shows the scatter plots for the needed CPU-time and the memory usage. Shown are the tasks that are solved correctly by both analyses. Despite the fact that DCPA↓ with the *network connection* solved 34 tasks less than DCPA↓ with the *in-memory connection*, Figure 28 shows a slight time-loss and a higher memory usage when communicating over the network. The additional threads for receiving messages and the process of sending large messages consumes time that is not needed with the *in-memory connection*.

6.2.9 Distributed Fault Localization

We already showed the theoretical performance boost for the distributed fault localization. Unfortunately, we do not have access to a reliable benchmark set for fault localization tasks to (dis)prove the assumed effect. Still, we can prove the soundness of the distributed fault localization. We compare the result of both approaches with the help of a set of hand-crafted tasks. The set

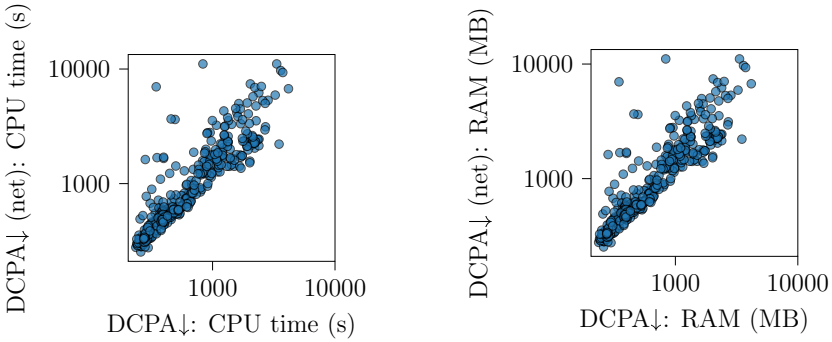


Figure 28: Comparison of the CPU time and the memory usage for the *in-memory connection* and the *network connection*.

Table 20: Error-prone lines according to fault localization.

task	name	distributed FL	FL
1	prime factor	12, 35	12, 35, 36
2	fault	4	4, 6, 12, 13
3	ifs	14	12, 14
4	simple	14	14, 15
5	prime	29	27, 28, 29
6	gcd	timeout	24, 34, 36

consists of six tasks with easy to complex problems. The tasks can be found here[3]. We enable the option *flPreconditionAlwaysTrue* for the distributed analysis. Table 20 shows the evaluation of the six tasks. The first two columns contain the ID of the task and its name. Columns 3 and 4 list the outputted lines where adaptions may fix the bug in the given program. For the first five tasks the distributed fault localization returns at least one fault reported by the existing fault localization technique. Since the undistributed implementation returns every possible irreducible subset, it often reports more error locations than our distributed analysis. However, for task 6, the distributed approach does not find a solution because the verification does not terminate in time. Unfortunately, there are many tasks with timeouts. Subsequently, the evaluation of the fault localization is not yet feasible for larger programs where we could compare the duration of fault localization. Although the evaluation only concentrates on a small set of programs, these experimental results give a first insight in the correct application of fault

[3] `https://doi.org/10.5281/zenodo.6224978`

localization in our distributed framework.

6.3 Discussion

6.3.1 Disadvantages

Currently, our approach suffers from a great number of satisfiability checks in the backwards analysis and large messages (around 200,000 characters on average at an advanced stage). The (de)serialization of the *pointer target set*, containing information about pointers of, e.g., arrays and structs is not supported yet. As soon as the support for the (de)serialization is added, the number of correctly solved tasks should increase. Moreover, the missing support for abstraction causes the path formulas to grow large in size increasing the time to check the formulas for satisfiability. Despite the implementation of the GivenSizeDecomposition, our implementation still has to deal with a larger number of workers. That also increases the number of spawned threads and, hence, they do not really run parallel. Subsequently, the wall time increases because we process to the verification result unimportant messages first.

6.3.2 Advantages

Disregarding the missing support of the (de)serialization of the *pointer target set* and the abstraction, theoretically, our approach should need less time to find a proof and equal time to find a violation. Proofs are found by declining all *error condition* messages. Since we reuse the information of the forward analysis, analyses from two directions contribute to finding a proof. Other workers might already have gained all the necessary knowledge to disprove the reachability of error locations as the SAT-check of the *proceed operator* deems it unsatisfiable. Contrary, *error condition* messages always have to be propagated to the *root worker* to be recognized as violation. Beyond that, the number of SAT-checks grows if a worker with many predecessors propagates the *error condition* to the root. Hence, we do not expect an improvement in theory. Figure 29 illustrates the concurrent work, leading to a faster proof. The decomposition algorithm produces two blocks (the figure does not show the root worker) and spawns two workers (left-hand side of the figure). Initially, both workers run a forward analysis but *Worker 2* finds an error location and runs a backward analysis instead. This happens simultaneously (middle part of the figure). *Worker 1* does not find a violation and broadcasts the path formula to *Worker 2*. Now, *Worker 1* runs the

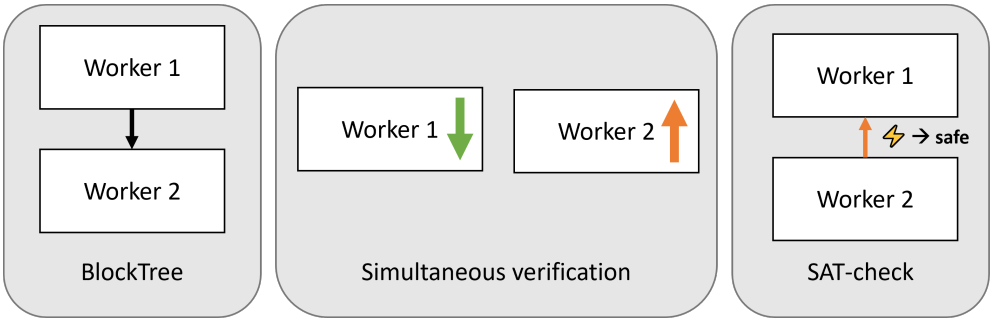


Figure 29: Parallel contribution of workers to a proof

backward DCPA since it receives a message of type ErrorCondition. The *proceed operator* checks the recently computed path formula of the forward analysis and the error condition from *Worker 2* for satisfiability (right-hand side of the figure). The solver proves the conjunction unsatisfiable and the *result worker* broadcasts the verification result True by using the results of the backward and the forward analysis.

6.3.3 Further Improvements

A possible optimization prevents workers to send duplicate ErrorCondition messages if the program under analysis is loop-free. In case no loops exist, identical ErrorCondition messages provoke the exact same computational steps for all predecessors and thus can be omitted, too. There exist cases for programs with loops where we wrongfully decline an *error condition* because the loop is not unrolled sufficiently. Therefore, we cannot discard them, as the precondition of the workers becomes stronger by another iteration.

Finally, we expect to raise the number of correctly solved tasks with the support of abstraction and the *pointer target set* as mentioned before.

6.3.4 Summary

Our approach and the implementation are sound and produce correct verification results. However, the performance suffers from the high number of sent messages. More resources cause workers to issue more messages which in return costs more time to process again. The reduction of messages containing irrelevant information is just one possible entry point for future improvements. Additionally, the evaluation reveals more points of departure for boosting the performance, giving us confidence that the approach might scale

well. We already present some optimizations that have an measurable impact on reduction of resources and with the support of abstraction, we hope to make a fist step towards a competitive implementation of the presented distributed approach. Moreover, the fact that distributing the verification to multiple workers performs significantly better than executing it on one blocks encourages us to further explore the possibilities of this approach.

6.3.5 Threats to Validity

We run the experiments on the benchmark set of SV-Comp 2022 which currently is the biggest collection of verification tasks. However, there is always the risk biased benchmark sets.

Our implementation may contain undetected bugs. Nevertheless, the experimental results are sound and comply with our assumptions reinforcing our belief that the implementation works as intended.

Additionally, the order of processing messages is nondeterministic. Depending on when the operating system schedules workers, the duration of the analysis varies. We implement the message prioritization to mitigate the nondeterminism to at least some extend.

Benchmarks may be invalid because of varying initial conditions. However, we use BenchExec, a tool for reliable benchmarks, for our evaluation. BenchExec is also used to run the benchmarks for SV-COMP [1]. Furthermore, all benchmarks are executed on the same machines.

The evaluation of the fault localization tasks is based on a selection of hand-crafted tasks. Therefore, the results might differ for currently uncovered cases. However, the hand-crafted tasks are complex enough to indicate whether the distributed fault localization is actually working.

6.4 Future Work

6.4.1 Combining Analyses and Blocks

Our approach allows a variety of future adaptions and improvements. The flexibility of the actor model permits the introduction of other workers that may or may not operate on the same blocks. Since the processing of a message solely relies on the target CFA node, we can even spawn workers working on a differently partitioned CFA. Messages for specific target nodes can subsequently be used by all blocks starting or ending with that target node. Another possible scenario may run different distributed CPAs on the same blocks and the verification results of the one analysis strengthen the abstract

states of the other analysis. In case, either one of the CPAs proves a violation itself, we can report that the program is unsafe. For this, we additionally have to implement more distributed CPAs. The value analysis. for example, can easily be implemented as distributed CPA as we only have to communicate the current map of variables to their values. The combine operator inherits from the merge operator and we stop if values at the same location do not match. Furthermore, we did not yet take a look at distributed CPAs where every worker executes either the forward or the backward analysis but not both. Running the actual CPA on the whole CFA provides us with an upper bound. No matter what, we take at most as long as the parent CPA. The implementation of the *network connection* allows running completely different instances of CPAchecker on different machines enabling us to use even more CPU power. The objective becomes to reduce the wall-time to the expense of the CPU-time. Once again, this enlarges the number of possible extensions. The way we implement the messages allows to change the protocol at any time just by introducing new message types.

We can optimize our free resources by deactivating unnecessary workers. If a block contains an unsatisfiable formula after the forward analysis, we can shutdown all workers analyzing successor blocks since they analyze unreachable blocks. This frees resources and memory because fewer threads require the CPU and large messages can be deleted. In addition, we can experiment with deactivating the forward or the backward analysis and solely rely on one of the analyses reducing the number of expensive solver calls and the number of memory-heavy messages.

6.4.2 Port Blocks to BAM-Blocks

In Section 2.1 we describe BAM which already is implemented in CPAchecker. BAM also partitions the CFA into blocks and a data structure for these blocks already exists. In the future, our blocks should extend these blocks. The advantage is, that we can make use of already implemented visualization techniques and other utility methods.

6.4.3 Strategy Selection

CPAchecker comes with a strategy selection analysis [5], meaning that based on some features of a program the probably best algorithm for verifying the given task will be chosen. In our approach, we could run a strategy selection for every block and run the best algorithm accordingly. However, this comes with a challenge. Every worker has to understand messages that do not represent abstract states of the CPA of the worker. There are two

ways to tackle this problem: either we extend distributed CPAs in a way that they know how to serialize messages for other CPAs or they know how to deserialize messages of other CPAs. Since many of the CPAs rely on path formulas, the effort to achieve this may be manageable. The value analysis, for example could simple convert the variable-value map to a path formula ($\{x : 0\} \Rightarrow x^{\langle 0 \rangle} = 0$). The strategy selection benefits from intelligent block finding. CPAchecker implements techniques capable of calculating invariants for loops. If we, for example, choose our blocks in a way that every loop maps to one block, we might be able to speed up the analysis significantly.

6.4.4 Abstraction

Our implementation already supports abstraction. Unfortunately, the needed abstraction strategy Elimination causes problems with the underlying solver used by CPAchecker. The idea of the *elimination abstraction* is, to only track variables that are really needed to reach the error and eliminate all other variables from the formulas to make messages smaller in size. As soon as Elimination works, we can experiment with it. Another goal for the future is to make our approach work with CEGAR [9, 12].

6.4.5 Other Improvements

The visualization of blocks is not perfect yet. There are many possible ways to extend it. An overview of when which worker entered and finished an analysis would also be helpful for debugging. On this way, we can identify blocks that are stuck or take long. The insights help to improve the code as we are able to understand and recognize such problems faster.

Another minor improvement concerns the error handling. Workers that fail to send messages are believed to be working. There is no way to reach them. For debugging purposes it is beneficial to better log the errors and to shutdown workers after some time. Currently, this is not implemented as we would have to spawn even more threads.

The reached set in the main analysis is not yet adapted accordingly. Subsequently, the graphical representation of the verification cannot be shown. In the future this should be possible as well. This is a non-trivial task because every computed abstract state has to be transferred to the main thread and many abstract states are computed multiple times because updates to the pre- and post-conditions might trigger the same analysis twice.

For the predicate analysis, we can implement a combine operator that combines multiple abstract states to one abstract state by calculating the

disjunction of the given path formulas. There may be analyses where this is not possible. For such cases, we can extend the combine operator to return a set of abstract states that are added to the waitinglist and the reached set initially. Successive blocks have to deal with every abstract state separately.

7 Conclusion

In this work, we introduce the concept of DCPAs, allowing the distribution of arbitrary existing CPAs to threads by extending it with four operators. With the help of forward and backward analyses, error location are proven (un)reachable distributed over all workers. The workers contribute to the verification result by running analyses, regardless of the progress of all other workers.

To evaluate and explore this approach, we provide the implementation of a framework supporting distributed DCPAs. Our framework is integrated in the already highly configurable CPAchecker. For existing CPAs in CPAchecker it suffices to implement the four operators *serialize, deserialize, combine* and *proceed* and our framework automatically runs the CPA distributed on the *analysis workers* if activated. Currently, the most challenging part for integrating DCPAs is the implementation of the *(de)serialize operators* since it requires to represent potentially complex data structures as strings. Nevertheless, to make use of the distributed approach, users do not need to take care of anything but the implementation of the operators.

The *analysis worker* run DCPAs on *code blocks* obtained by various decomposition algorithms. We provide two implementations for decomposing CFAs: the *linear* and the *given size decomposition* with both having advantages and disadvantages. Whereas the GivenSizeDecomposition reduces the number of workers, the LinearDecomposition is easily extensible to other approaches and is able to solve more tasks. Moreover, our approach allows an easy integration of other concepts as, for example, fault localization.

Currently, the performance of our approach is limited by the available hardware resources and the potential execution of equal computation steps, increasing both, the needed time and the needed memory. However, the evaluation gives insightful hints for further improvements. We already achieve a performance boost by reducing the number of SAT-checks and we expect further improvements as soon as abstraction enriches our approach. The reduction of the size of messages and the support of the *pointer target set* will

also increase the number of correct results.

In summary, we make a first step towards distributed analyses of programs. With the framework as foundation, we have a multitude of possibilities for further adaptions and extensions as well as experiments with a variety of other existing approaches as, e.g., strategy selection.

Bibliography

[1] D. Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 887–904. Springer, 2016.

[2] D. Beyer. Software verification: 10th comparative evaluation (sv-comp 2021). *Tools and Algorithms for the Construction and Analysis of Systems*, 12652:401, 2021.

[3] D. Beyer. Progress on software verification: SV-COMP 2022. In *Proc. TACAS (2)*, LNCS 13244. Springer, 2022.

[4] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *2009 Formal Methods in Computer-Aided Design*, pages 25–32. IEEE, 2009.

[5] D. Beyer and M. Dangl. Strategy selection for software verification based on boolean features. In *International Symposium on Leveraging Applications of Formal Methods*, pages 144–159. Springer, 2018.

[6] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *International Conference on Computer Aided Verification*, pages 504–518. Springer, 2007.

[7] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.

[8] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Formal Methods in Computer Aided Design*, pages 189–197. IEEE, 2010.

[9] D. Beyer and S. Löwe. Explicit-state software model checking based on cegar and interpolation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 146–162. Springer, 2013.

[10] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. 2003.

[11] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The mathsat5 smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[12] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[13] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[14] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[15] C. Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[16] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. *ACM SIGPLAN Notices*, 46(6):437–446, 2011.

[17] M. Kettl. Fault localization for formal verification. an implementation and evaluation of algorithms based on error invariants and unsat-cores. Bachelor's Thesis, LMU Munich, Software Systems Lab, 2020.

[18] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[19] P. O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.

[20] R. Qiu, S. Khurshid, C. S. Pasareanu, and G. Yang. A synergistic approach for distributed symbolic execution using test ranges. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 130–132. IEEE, 2017.

[21] P. Wendler. Reliable benchmarking: Requirements and solutions. 2019.

[22] D. Wonisch and H. Wehrheim. Predicate analysis with block-abstraction memoization. In *International Conference on Formal Engineering Methods*, pages 332–347. Springer, 2012.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Kettl Matthias

Matthias Kettl

22.02.2022

Datum