DEPARTMENT OF INFORMATICS

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Computer Science

Enhancing Distributed Summary Synthesis with Data-Flow Analysis

Sara Ruckstuhl

Supervisor:	Prof. Dr. Dirk Beyer
Mentor:	Matthias Kettl
Submission Date:	04.11.2024

I confirm that this bachelor's thesis in computer science is my own work and I have documented all sources and material used. I used ChatGPT to help with rephrasing single sentences and small paragraphs.

Munich, 04.11.2024 Sara Ruckstuhl

Acknowledgments

I would like to give a big thank you to everyone who has supported me throughout the process of writing my thesis. I'm especially grateful to my mentor, Matthias Kettl, whose advice and readyness to answer all my questions helped me immensely.

I am also grateful to my supervisor, Prof. Dr. Dirk Beyer, for providing me with the opportunity write my thesis at this chair. Finally, thanks to my friends and family, whose encouragement and understanding helped me stay motivated throughout the journey.

Abstract

With increasing software complexity, scalable and precise verification is essential, especially in safety-critical areas. Distributed Summary Synthesis (DSS) supports scalability by enabling parallel processing of program segments (blocks). However, it faces limitations in achieving early-stage abstraction due to the inherent laziness of Predicate Analysis, which only refines abstractions when errors are detected. This thesis addresses this by integrating Data-Flow Analysis (DFA) into DSS, enhancing the initial information shared among program blocks to potentially accelerate and improve verification. Implemented in CPAchecker, DFA runs in parallel with Predicate Analysis, providing coarse summaries that strengthen the preconditions for successor blocks. Experimental evaluation using SV-COMP 2024 benchmarks, however, indicated that while DFA integration occasionally improved verification coverage, it also introduced additional resource demands. This increase in CPU time, wall time and memory usage, due to message handling and serialization and deserialization overhead, limited the number of programs that could be verified compared to the DSS implementation with only predicate analysis. This trade-off suggests that additional optimizations are needed to reduce performance costs and better harness the potential of DFA for scalable and effective verification.

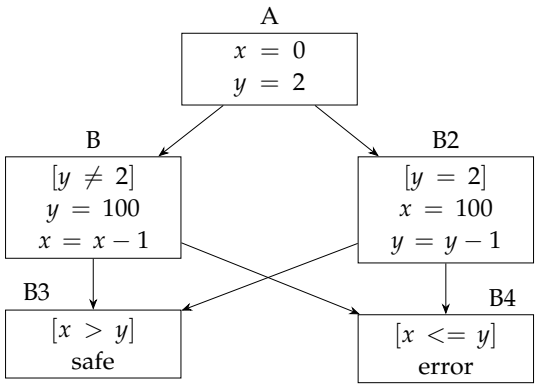
Contents

1 Introduction

In today's digital world, software systems are integral to daily life, making their reliability and correctness, especially in safety-critical areas, crucial. Software verification ensures that software behaves as intended, preventing costly errors or catastrophic failures [10].
As software systems grow larger and more complex, scalability becomes a major challenge for verification. Distributed Summary Synthesis (DSS) [5] addresses this by dividing the program into smaller, manageable units, or blocks, and verifying each block independently. Each block functions as a separate verification task, with its own precondition, postcondition, and a block summary. This division enables parallel processing, where each block can be verified on separate workers.
In DSS, the analysis operates in two directions: top-down and bottom-up. In the top-down approach, preconditions are refined as each block propagates its postcondition (summary of its verified state at exit) to its successor blocks, providing them with information that helps them establish more accurate preconditions. Conversely, in the bottom-up approach, if a block reaches a violation state that contradicts the program's safety specifications, it computes a "violation condition" at its entry point, which is then sent back to predecessor blocks. If this violation condition propagates all the way back to the program entry, an unresolvable error is confirmed, indicating the program is unsafe. If all blocks complete their analysis without forwarding a violation, the program is deemed safe.
The current DSS implementation relies primarily on Predicate Analysis to verify each block. However, this approach has limitations. Predicate Analysis is thorough but lazy, as it only refines summaries when it detects an error condition. Consequently, blocks without errors pass only trivial information to their successors, which slows down verification in the early stages.
This thesis explores an enhancement to DSS by integrating Data-flow Analysis (DFA) alongside Predicate Analysis. DFA provides quick, coarse summaries of each block's state, supplying initial information about variable values and constraints. At the start of the verification process, DFA runs on each block in parallel with Predicate Analysis. These summaries are then shared with successor blocks, which can use this information to strengthen their preconditions. This integration potentially accelerates verification by enabling each block with predecessors to begin with useful starting information.

```
1  int main() {
2     int x = 0;
3     int y = 2;
4     if (y == 2) {
5        x = 100;
6        y--;
7     } else {
8        y = 100;
9        x--;
10    }
11    assert(x > y);
12 }
```

(a) Example program in C



(b) Example block graph

Figure 1.1: Program with example decomposition into five blocks

The example in Figure 1.1 illustrates this process with a simple program divided into five blocks. DFA is run on each block to provide quick, initial summaries. For instance, when DFA runs on block A, it determines that at block exit, $x = 0$ and $y = 2$. This information is then passed to blocks B and C, allowing them to strengthen their preconditions. Blocks B and C now assume $x = 0$ and $y = 2$ at their entry points. In parallel, Predicate Analysis checks each block for potential error states. In this example, only block E reaches an error state, prompting it to compute a violation condition at its entry location, which it then sends to its predecessor blocks, B and C. In the next iteration, Predicate Analysis is run again on B and C with their updated precondition and the communicated violation condition. Given that $y = 2$, it becomes clear that block C is not reachable because the condition $y \neq 2$ is not satisfied. Conversely, block B is reachable since $y = 2$ holds true, and Predicate Analysis confirms that the assertion $y < x$ is not violated at the block exit. This combined analysis allows us to conclude that the entire program is safe, but with fewer iterations compared to using Predicate Analysis alone.

The primary contribution of this work is the extension of Data-Flow Analysis (DFA) to a distributed analysis within DSS. To enable DFA to communicate across blocks, we implemented a serialization and deserialization process for its results, which allows them to be transmitted efficiently between blocks. Additionally, we integrated this distributed DFA into the existing DSS implementation, running it with Predicate Analysis to provide quicker, coarse summaries at the beginning of the analysis.

We evaluated the extended DSS implementation by comparing it to the current DSS implementation on a set of benchmark programs. The results revealed that, while some programs benefitted from the DFA summaries and were verified in less time, the addi-

tional serialization and deserialization of DFA results increased resource consumption. This added overhead led to fewer programs being correctly verified, showing that while DFA provides the analysis with more information, it also introduces challenges that negatively impact the overall verification performance.

2 Related Work

2.1 Adjustable Block-Encoding (ABE)

ABE [1] builds on existing techniques for software model checking, such as Single-Block Encoding (SBE) and Large-Block Encoding (LBE) [6], by providing a flexible framework that allows for adjustments between these two approaches and beyond. In SBE the predicate abstraction is computed after every single program operation, leading to a large number of expensive theorem prover calls. In LBE the abstraction is computed at function calls or at head locations of loops. LBE therefore already significantly reduces the number of abstraction steps by combining multiple operations into a single block. ABE unifies these two approaches by introducing the block-adjustment operator, which allows for flexible configuration of the number of operations encoded in one formula per abstraction step. ABE has shown to be more efficient than SBE and LBE, as it can adapt to the program structure and the desired level of precision.

Both, DSS and ABE share the fundamental idea computing an abstraction only at certain points in the program. Additionally, unlike ABE, which analyses blocks in a hierarchical manner, DSS emphasizes parallelism. Each block's summary can be computed immediately, without waiting for predecessor blocks to complete their analysis.

2.2 Block-abstraction Memoization (BAM)

BAM [12] divides the CFA of a program into blocks, which may represent bodies of functions, nested loops or other repetitive structures that can be encountered multiple times throughout execution. The partition of the CFA is required to be either nested or disjoint, ensuring that blocks either do not overlap or one is fully contained within another. For each block, BAM computes an abstract reachability tree (ART) which can be reused in future executions of the same block. During the analysis of a block, BAM weakens the abstract state by reducing it to only the variables relevant to the current block, thus minimizing complexity. Afterwards, the abstract state is strengthened by restoring the information about the variables outside of the block.

Similar to ABE, BAM also supports the sequential analysis of blocks with a predecessor-

successor relationship. Additionally, blocks that are in no direct predecessor-successor relationship can be processed simultaneously, which allows for some degree of parallelism.

2.3 Combining Data-flow Analysis with Predicate Abstraction

There already exists research, which has explored the combination of data-flow analysis with predicate abstraction to improve the efficiency and precision of software verification. Traditional predicate abstraction techniques, although powerful, can be computationally expensive due to the number of iterations needed to refine the abstraction and eliminate spurious counterexamples. Data-flow analysis, on the other hand, provides a more scalable solution but often lacks the precision required for path-sensitive verification.

"Predicated lattices" [7] is a concept which enhances traditional data-flow analysis by incorporating predicates. This technique partitions the program state using a set of predicates and tracks different lattice elements for each partition. By integrating data-flow facts with predicates, this approach achieves greater precision than standard data-flow analysis. The key benefit lies in its ability to reduce false positives by avoiding imprecisions that arise from infeasible paths. This method has shown significant improvements in verification times, as fewer iterations are required compared to purely predicate-based analyses.

Statically computed invariants can be used to strengthen the transition relation at different program locations [9]. Their approach introduces octagonal invariants into the predicate abstraction and refinement loop, which helps reduce spurious counterexamples and the number of refinement iterations. By efficiently discovering invariants at each program location, the abstract model becomes more precise, leading to fewer refinement steps and reduced computational overhead.
This thesis explores the idea of generating stronger preconditions by using the information of data-flow analysis to improve the efficiency of the predicate abstraction within a block.

3 Background

3.1 Control Flow Automaton (CFA)

A CFA is a directed graph which shows the control flow of a program. The nodes of the automaton represent the different locations in the program and the edges represent transitions.

Definition 1 (CFA). A CFA is formally defined as a triple $CFA = (L, l_0, G)$, where L represents a finite set of program locations and $l_0 \in L$ is the start location marking the entry point of the program being analysed. The set G comprises all possible edges, with each edge $g = (l, o, l') \in G$ being an element of $L \times Op \times L$. This indicates that executing operation o at location l transitions the program to location l'. For the sake of simplicity, we assume that an operation can either be an assumption, such as $[y <= 3]$, or a variable assignment, like $x = 0$.

Figure 3.1 shows a CFA which has been constructed from the given example program. The start node l_0 represents the program's entry point. After performing the initial operation $x = 0$, the program counter moves to the subsequent node in the CFA. When a conditional statement is encountered, there are two possible paths, resulting in a branching at l_2: either $y \neq 2$ or $y = 2$.

3.2 Configurable Program Analysis (CPA)

Definition 2 (CPA). A CPA [4] $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$ comprises an abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator merge, and a termination check stop.

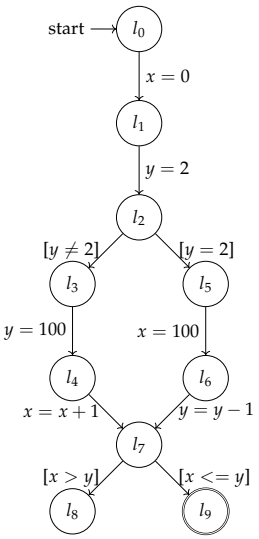
The next section explains each of these four components of a CPA in more detail.

3.2.1 Components of a CPA

Abstract Domain
The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set C of concrete states, a semi-lattice $\mathcal{E}$, and a concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{E} = (E, \sqcup, \top, \sqsubseteq)$ includes a set E of elements, a top element $\top \in E$, a partial order $\sqsubseteq \subseteq E \times E$, and a join operator

```
1  int main() {
2     int x = 0;
3     int y = 2;
4     if (y == 2) {
5        x = 100;
6        y--;
7     } else {
8        y = 100;
9        x--;
10    }
11    assert(x > y);
12 }
```

(a) Example program in C



(b) Example CFA

Figure 3.1: Example program with corresponding CFA

$\sqcup : E \times E \to E$. The elements of E are the abstract states. The concretization function $[\![\cdot]\!] : E \to 2^C$ maps each abstract state to the set of concrete states it represents.

Transfer Relation
The transfer relation $\rightsquigarrow \subseteq E \times G \times E$ maps each abstract state e to potential new abstract states e' as abstract successors of e. Each transfer is labeled with a control-flow edge g. We write $e \overset{g}{\rightsquigarrow} e'$ if $(e, g, e') \in \rightsquigarrow$, and $e \rightsquigarrow e'$ if a g exists such that $e \overset{g}{\rightsquigarrow} e'$.

Merge Operator
The merge operator merge : $E \times E \to E$ combines the information of two abstract states into one. The result of $merge(e, e')$ can range between $e' \leq e \leq \top$, meaning the merge operator weakens the second abstract state e' based on the first e. If we do not want to merge, we use $\text{merge}^{sep}(e, e') = e'$; if the merge operator should have the same effect as the join operator we use $\text{merge}^{join}(e, e') = e \sqcup e'$.

Stop Operator
The stop operator stop : $E \times 2^E \to \mathbb{B}$ takes an abstract state e and a set of reached abstract states R. Its configuration determines wether to continue with the exploration of the abstract state e. In this work we use the $\text{stop}^{sep}(e, R)$ operator, which checks if there exists an abstract state e' in the set of reached states R such that e' subsumes the

Algorithm 1 CPA Algorithm [4]

Input: A configurable program analysis $D = (D, \rightsquigarrow, \text{merge}, \text{stop})$ and an initial abstract state $e_0 \in E$

Output: a set of reachable abstract states

Variables: a set reached of elements of E, a set waitlist of elements of E

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

current state e. If such a state exists, the analysis can stop exploring e further, as its behaviour has already been captured by e'.

3.2.2 CPA Algorithm

The CPA algorithm (Algorithm 1) operates by initializing both a waitlist and a reached set with the initial abstract state e_0. The algorithm continues processing as long as there are elements in the waitlist. In each iteration, an element is removed from the waitlist (Line 4), and its abstract successors are computed by using the transfer relation. In Line 7, the algorithm attempts to merge each successor successor e' with any corresponding element already present in the reached set. If the merge results in a new element, the original element in the reached set is replaced by the merged result (Line 10).

In Line 13, the stop operator is invoked to decide if the newly computed successor should be added to the waitlist for further exploration. This process repeats until the waitlist is empty, signifying that all reachable abstract states have been fully explored. The algorithm then returns the set of reached abstract states.

3.2.3 Composite Program Analysis

A Composite CPA is a program analysis, which allows different CPAs to be combined within one CPA and is formally defined as $\mathcal{C} = (\mathbb{D}_1, \mathbb{D}_2, \rightsquigarrow_\times, \text{merge}_\times, \text{stop}_\times)$ and consists of two CPAs $\mathbb{D}_1$ and $\mathbb{D}_2$, sharing the same set of concrete states $\mathcal{C}$. The composite operators $\rightsquigarrow_\times$, $\text{merge}_\times$, and $\text{stop}_\times$ are derived from the corresponding components of both CPAs and manage how information from both analyses is combined without directly manipulating the internal details of the individual CPAs.
The composite CPA shares a set of concrete states $\mathcal{C}$, with E_1 and E_2 representing the abstract states of the individual analyses. The product domain $D_\times = D_1 \times D_2$ is computed by taking the direct product of the two domains. It operates on the product lattice which is constructed from the abstract states of each CPA. To ensure that the composite analysis represents all concrete states that are valid in both analyses, the concretization function $[\![\cdot]\!]_\times$ is defined as the intersection of the concretization functions of the individual analyses.

Strengthening Operator
The strengthening operator $\downarrow$: $E_1 \times E_2 \to E_1$ refines the abstract state of the domain D_1 of the first CPA. It does so by taking an abstract state $e_1 \in E_1$ and utilizes information from an abstract state $e_2 \in E_2$ of another CPA to further refine e_1. The resulting abstract state $\downarrow (e_1, e_2)$ represents a more constrained set of concrete states than the original e_1, i.e., $[\![\downarrow (e_1, e_2)]\!] \sqsubseteq [\![e_1]\!]$.

3.2.4 Data-flow Analysis (DFA)

DFA is a technique which tracks the flow of data through a program's control flow graph. Typically, the goal is to gather coarse information about variable values at various points in the program. In this work, we focus on a specific form of DFA where the analysis targets intervals over program variables. Formally, this DFA can be described as a CPA with dynamic precision adjustment (CPA+) [3]. CPA+ extends the standard CPA framework by introducing a set of precisions and a precision adjustment function. In the context of CPA, precision refers to a finite set of predicates that controls the coarseness of the over-approximation of the abstract states. The CPA used in this CPA+ framework is the interval CPA $\mathbb{I}$, where the abstract domain consists of intervals

over program variables. In the following, we will elaborate on the components of this interval CPA+ $\mathbb{I}$, including the abstract domain, transfer relation, merge operator, and stop operator, as well as the precision adjustment.

Abstract Domain
The abstract domain in the Invariants CPA is based on expressions over intervals. Each abstract state is represented as a mapping function $M : X \to Expr$, where X is the set of program variables and $Expr$ is the set of arithmetic expressions. Since we work with the C programming language, the operators follow their C semantics, meaning that operators like bitwise shifts $\ll$ and $\gg$ exhibit behaviour according to the rules of C, such as shifting beyond the width of the data type. $Expr$ can include binary expressions, unary expressions, program variables and disjunctions of intervals and is recursively defined as $Expr \subseteq ((Expr \times B \times Expr) \cup (U \times Expr) \cup X \cup I)$. The set of supported binary operators B consists of $+, *, /, \%, =, <, \wedge, |, \vee, \&, \wedge, \ll, \gg, \cup$ and the set of supported unary operators U consists of $\neg, \sim, -$. I is the set of disjunctions of intervals which are represented as $[u, l]$ with $u, l \in \mathbb{Z} \cup \{\infty\}$. For instance, an abstract state could map an integer variable x to an interval like $x \to [1, 10]$, indicating that the variable x can take any value within range between 1 and 10.

Transfer Relation
The transfer relation of the Invariants CPA is responsible for updating the abstract state of the program variables as the program transitions from one location to another through the control flow edges, which can be either assignments or assumptions. The transfer relation applies these operations to modify the abstract state mappings accordingly. In the case of an assignment like $x := x + 1$, the transfer relation updates the current abstract state for x by adjusting its expressions, such as from $x \to [1, 10]$ to $x \to [2, 11]$. Similarly, for assume edges, the transfer relation constrains the abstract state based on the condition specified in the assume statement. For instance, if the assume statement is $x > 4$ and the current abstract state is $x \to [1, 10]$, the abstract successor state computed by the transfer relation will be $x \to [5, 10]$.

Precision
The precision used in the CPA+ $\mathbb{I}$ is defined as a triple $\pi = (Y, n, w)$ where Y is a subset of program variables which are labeled as important, n is the maximum nesting depth of expressions. A higher nesting depth allows for a more precise representation of the relations between the variables. Finally, w is a boolean indicating wether widening should be applied. Each abstract state e is paired with a precision π.

Merge Operator
The merge operator of CPA+ $\mathbb{I}$ an perform two key strategies, depending on the precision of the analysis: union and widening. Two abstract states are merged if their interval expressions match over all important variables. When merging occurs, the union of the interval expressions is taken. This means that for each variable, the resulting state will represent the combined range of values from both abstract states. For instance, if one state maps $x \to [1, 5]$ and another maps $x \to [6, 10]$, the merged state will map $x \to [1, 10]$. The widening operation further relaxes the abstraction by assigning a single, overapproximated interval to a variable by computing the upper and lower bounds of the intervals of the two states. For example, if a variable x was mapped to $[1, 10]$ in one state and [11,20] in another, after widening x might be mapped to $[1, \infty]$ ensuring that further refinement stops and the analysis can terminate.

Stop Operator
In the context of CPA+ $\mathbb{I}$ the stop^{sep} operator is used. This operator checks wether there exists an abstract state $e' \in R$ from the set of previously reached states R that subsumes the current state e. If such a state exists, the exploration of e can stop, as it is already represented by e' and therefore the current state is not added to the waitlist.

Precision Adjustment Function
The precision adjustment function of CPA+ $\mathbb{I}$ dynamically refines or relaxes the precision during the analysis. In can modify the set of important variables, adjust the allowed formula nesting depth, or disable widening. For example, if the analysis finds an error path, the precision can be refined by adding all the variables that occur in assumes to the set of important variables.

3.2.5 Predicate Analysis

The Predicate CPA $\mathbb{P}$ uses predicate abstraction to represent program states. It can also be defined as a CPA+ $\mathbb{P} = (\mathbb{P}, \pi, \rightsquigarrow, \text{merge}, \text{stop})$ where $\pi \subset \Pi$ denotes the set of precisions which are used to construct abstract states. This is achieved by overapproximating concrete states. This section describes the components of Predicate CPA [1], including the abstract domain, transfer relation, merge operator, and stop operator.

Abstract Domain
The abstract domain of Predicate CPA is defined by sets of predicates, each of which describes properties of the concrete states. Due to the use of adjustable-block encoding (ABE), where predicate abstraction is only computed at specific program locations, the abstract state e is defined as the four-tuple $(l, \psi, l_\psi, \varphi)$. Here, l represents the current pro-

gram location, ψ is the abstraction formula, which is a combination of predicates from the precision ϕ, l_ψ refers to the location where the abstraction was last computed. Path formula φ holds the path conditions leading to the abstract state. The semi-lattice used in the Predicate Analysis is defined as $\mathcal{E} = (E, \sqcup, \top, \sqsubseteq)$, where $\top = (l_\top, true, l_\top, true)$ represents the top element of all abstract states from E. The partial order $\sqsubseteq$ is defined as $(l_1, \psi_1, l_{\psi_1}, \varphi_1) \sqsubseteq (l_2, \psi_2, l_{\psi_2}, \varphi_2)$ if $(e_2 = \top) \vee ((l_1 = l_2) \wedge (\psi_1 \wedge \varphi \implies \psi_2 \wedge \phi_2))$. The join operator $\sqcup$ computes the least upper bound of two abstract states based on the partial order.

Transfer Relation
The transfer relation defines how abstract states are updated as the program transitions from one location to another through a control-flow edge $g = (l_1, o, l_2) \in G$. It contains all triples with (e, g, e') where $e = (l_1, \psi_1, l_\psi, \varphi_1)$ and $e' = (l_2, \psi_2, l_\psi, \varphi_2)$. The block-adjustment operator $blk(e, g)$ which maps a CFA edge to true or false, determines wether a computation of an abstract state is required. The strongest postcondition operator SP_o computes ψ_2 as the strongest boolean combination of predicates in ϕ after applying the operation o. In this case, the path formula is reset to $\varphi_2 = true$. If no abstraction is required, the path formula is updated, and the location of the last computed abstraction l_φ remains the same.

Merge Operator
In the case of two abstract states $(l_1, \psi_1, l_\psi, \varphi_1)$ and $(l_2, \psi_2, l_\psi, \varphi_2)$ sharing the same location $l_1 = l_2$, the same abstraction formula $\psi_1 = \psi_2$, and the same abstraction point l_ψ, their path formulas are combined using a disjunction $\varphi_1 \vee \varphi_2$. If these conditions are not satisfied, the second abstract state is taken as the result of the merge operation.

Stop Operator
The stop operator of the Predicate CPA checks wether the current state e is subsumed by any state already in the reached set R. If there exists a state e' in R that already covers e, meaning $e \sqsubseteq e'$ holds, the exploration of the current state is terminated.

Counterexample-Guided Abstraction Refinement (CEGAR)
CEGAR is a technique that can be used together with Predicate Analysis to refine the precision that define the abstraction. The analysis starts with a simplified, coarse abstraction and refines this abstraction based on the results of the analysis. When a violation of the property being verified is found, an error path is generated. This path represents the sequence of steps that lead to the violation.
If the error path corresponds to an actual error in the program, the analysis reports the violation. However, sometimes the error path does not represent a real error in

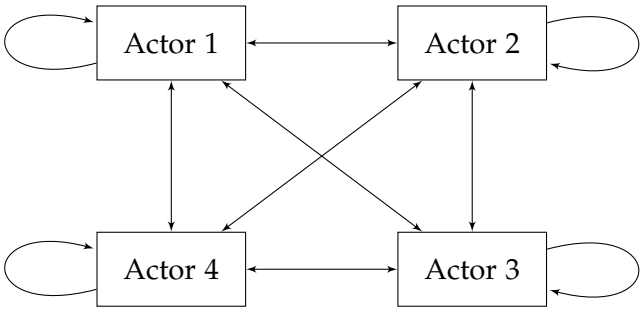


Figure 3.2: Actor model with four actors each communicating by broadcasting messages to every block within the model including itself

the program's execution. This is known as a spurious counterexample. A spurious counterexample arises when the abstraction is too coarse, meaning it simplifies the program so much that the error path which is found is not feasible. In such cases, the analysis refines the abstraction by adding more detail, ensuring that the same spurious path will not be explored again. This refinement process continues until the analysis either proves the absence of errors or finds an error path that is feasible.

3.3 Actor Model

The actor model [8] is a framework for concurrent computation where "actors" are independent units of computation that communicate through message passing. Each actor processes messages asynchronously and can modify its own state, send messages to other actors, and create new actors.
In the context of DSS, the actor model is used to implement distributed verification tasks, where each block of the program is handled by an actor that communicates its results via postcondition or violation condition messages to other actors. Figure 3.2 shows an example of an actor model with four actors, where the boxes represent the actors and the arrows specify the channels through which messages can be passed.

3.4 Distributed Summary Synthesis (DSS)

DSS [5] extends CPA by decomposing the program into smaller, independent verification tasks (blocks) and distributing these tasks across multiple actors, allowing for parallel verification.

3.4.1 Decomposition of CFA

Definition 3 (Block). Given a CFA $P = (L, l_0, G)$ a block can be viewed as a weakly connected subgraph $b = (L_b, L_{entry}, L_{exit}, G_b)$ with nodes $L_b \subseteq L$, edges $G_b \subseteq G$, an entry node l_{entry} and an exit node l_{exit}. Typically, none of the predecessors of L_{entry} and none of the successors of L_{exit} are included within L_b of block B. In case of a block covering a full loop, the loop head functions as both the entry and exit node of the block.

Definition 4 (Block-Adjustment Operator). The Block-Adjustment Operator, denoted as $blk : G \to \{true, false\}$, maps each edge of the CFA $g = (l, o, l')$ to either $true$ or $false$. If the operator returns $true$ for an edge, this indicates that the edge marks the end of the current block, and any outgoing edge of the target node l', begins a new block. Conversely, if the operator returns $false$, the edge is part of the current block. Two trivial concrete definitions of the Block-Adjustment Operator are blk^{sbe}, which always returns $true$, meaning each block consist of a single program operation, and blk^{false}, which always returns false leading to the entire program being treated as a single block.

Definition 5 (Block Graph). A Block Graph $\mathscr{B} = (B, G_{\mathscr{B}})$ is a directed Graph with a set of blocks B as nodes and a set of directed edges $G_{\mathscr{B}} \subset B \times B$ between them.

Definition 6 (Valid Decomposition of CFA). A decomposition of a CFA is valid if and only if for each node l of the resulting block graph $\mathscr{B}$ it holds that either l only appears in a single block or l is the node that connects block b with all its successor blocks. In that case l is the exit node of b and the entry node of each successor block of b.

3.4.2 Block Merging

Block merging is a strategy aimed at optimizing the number and size of blocks in the block graph. The objective is to reduce the number of blocks by alternately merging either horizontally or vertically until the number of blocks converges to a desired target number or until no further merges can be performed. The following defines the two types of block merges.

Definition 7 (Horizontal Merge). A horizontal merge can be performed when both the entry and exit location of two blocks are the same. Formally, consider two different blocks with identical entry and exit locations $b = (L_b, l_{\text{entry}}, l_{\text{exit}}, G_b)$ and $b' = (L_{b'}, l_{\text{entry}}, l_{\text{exit}}, G_{b'})$. The resulting block $b = (L_b \cup L_{b'}, l_{\text{entry}}, l_{\text{exit}}, G_b \cup G_{b'})$ is the horizontal merge of b and b'.

Definition 8 (Vertical Merge). A vertical merge of two blocks is possible when the exit location of one block matches the entry location of another block. Formally, given two blocks $b = (L_b, l_{\text{entry}}, l_{\text{exit}}, G_b)$ and $b' = (L_{b'}, l'_{\text{entry}}, l'_{\text{exit}}, G_{b'})$, a vertical merge is possible if $l_{\text{exit}} = l'_{\text{entry}}$. Also, for a vertical merge to be valid, block b must be the only block in the block graph with l_{exit} as its exit location, and block b' must be the only block with l'_{entry} as its entry location. The resulting block $b'' = (L_b \cup L_{b'}, l_{\text{entry}}, l'_{\text{exit}}, G_b \cup G_{b'})$ is the vertical merge of b and b'.

3.4.3 Message Passing in DSS

Definition 9 (Message). To enable communication between the blocks $b \in B$, we define messages M. A message $m \in M$ is a triple $T \times B \times C$, where $\tau \in T = \{\tau_{\text{post}}, \tau_{\text{vcond}}\}$ represents the type of the message. The type determines the content of the message: τ_{post} is used for transporting postconditions, which are sets of abstract states associated with a block, while τ_{vcond} is used for conveying violation conditions, which are sets of target-reaching states. The message also contains the actual content $\zeta \in C$, which can either be a postcondition or a violation condition, depending on the type τ, and $b \in B$ refers to the block from which the message is sent. Packed objects, such as sets of abstract states, are denoted with a subscript M. For example, $\{e\}_M$ indicates that the set $\{e\}$ has been packed into a message.

3.4.4 Distributed Configurable Program Analysis

A Distributed CPA $\mathcal{D} = (\mathbb{D}, \text{packPost}, \text{packVcond}, \text{unpackPost}, \text{unpackVcond})$ extends the conventional CPA by also defining how to pack abstract states E into messages and unpack messages into abstract states. The operator packPost : $2^E \times B \to 2^M$ is responsible for packing a set of abstract states which belong to block $b \in B$, into a set of postcondition messages. Conversely, the operator unpackPost : $2^M \times B \to 2^E$ unpacks a set of postcondition messages into abstract states. Similarly, the operator packVcond : $2^E \times B \to 2^M$ is used to pack target-reaching states into violation-condition messages. The counterpart operator unpackVcond : $2^M \times B \to 2^E$ unpacks violation-condition messages and restores the abstract states representing the target-reaching states.

Distributed Predicate CPA

The Distributed Predicate CPA functions by running predicate analysis with CEGAR on each blocknode of the distributed CPA. Predicate analysis is performed locally at each blocknode, deriving predicates using CEGAR to iteratively refine the abstraction at that block. This approach has been implemented in the current system, where DSS

Algorithm 2 DSS$(b, E_0^b, \varphi, \mathcal{A}, \mathcal{D})$ [5]

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

1: $T_\varphi := \{e \in E \mid e \not\models \varphi\}$
2: post $:= [(\tau_{\text{post}}, b', E_0^b)_M \mid b' \in B]$
3: vcond $:= [(\tau_{\text{vcond}}, b', \emptyset_M) \mid b' \in B]$
4: **while** true **do**
5: $\quad m :=$ nextMessage()
6: $\quad$ **if** $m = (\tau_{\text{post}}, b'_m, \cdot)$ **then**
7: $\quad\quad$ post $:= [(\tau_{\text{post}}, b', \cdot) \in \text{post} \mid b' \neq b'_m] \circ [m]$
8: $\quad$ **end if**
9: $\quad$ **if** $m = (\tau_{\text{vcond}}, b'_m, \cdot)$ **then**
10: $\quad\quad$ vcond $:= [(\tau_{\text{vcond}}, b', \cdot) \in \text{vcond} \mid b' \neq b'_m] \circ [m]$
11: $\quad$ **end if**
12: $\quad R_{\text{start}} := \{(e, \pi_0) \mid e \in \text{unpackPost}(\text{post}, b)\}$
13: $\quad T := \text{unpackVcond}(\text{vcond}, b) \cup T_\varphi$
14: $\quad R := \mathcal{A}_b(\mathbb{D}, R_{\text{start}}, R_{\text{start}}, T)$
15: $\quad E_R := \{e \mid (e, \cdot) \in R\}$
16: $\quad V := \{e \in E_R \mid [\![e]\!] \cap [\![T]\!] \neq \emptyset\}$
17: $\quad$ **if** $V \neq \emptyset$ **then**
18: $\quad\quad$ **broadcast** packVcond(V, E_R, b)
19: $\quad$ **else**
20: $\quad\quad$ **broadcast** packPost$(\{e \in E_R \mid e$ located at $l_{\text{exit}}\}, b)$
21: $\quad$ **end if**
22: **end while**

takes the Distributed Predicate CPA as an argument. The communication between blocknodes occurs via message passing. If the predicate analysis detects that the specification is violated, a violation-condition message is sent to predecessor blocks, signalling a potential error. Conversely, if the specification is satisfied at a blocknode, a postcondition message is sent to successor blocks to indicate that the analysis has successfully verified that portion of the program.

3.4.5 Distributed Summary Synthesis Algorithm

The DSS algorithm (Algorithm 2) operates by exchanging postconditions and violation conditions between the blocks to refine the analysis. The algorithm takes as input a

Algorithm 3 packPost$_A(E_{in}, b)$

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

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

Algorithm 4 packVcond$_A(V, E_R, b)$

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

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

Algorithm 5 unpackPost$_A(M_{post}, b)$

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

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

Algorithm 6 unpackVcond$_A(M_{vcond}, b)$

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

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

block b, its initial abstract states E_0^b, the specification φ, a reachability analysis $\mathcal{A}$, and a distributed CPA $\mathcal{D}$, which defines the packing and unpacking of messages.
The process begins by initializing the target states T_φ as all abstract states that violate the specification φ, along with empty lists for postconditions and violation-condition messages for each predecessor block b'. These lists will hold the most recent messages sent by other blocks.
The algorithm then enters a loop (Line 4) where it waits for the arrival of messages using the function nextMessage(). When a message $m = (\tau, b'_m, \cdot)$ arrives, the algorithm distinguishes between two types of messages. If the message is a postcondition message τ_{post}, the algorithm updates the list of postconditions by removing the old postcondition for block b'_m and adding the new postcondition (Line 7). If the message is a violation-condition message τ_{vcond}, the list of violation conditions is updated in the same way Line 10.

Table 3.1: Run of DSS with Predicate Analysis

	A	**B**	**C**	**D**	**E**
R_{start_0}	$\{pc = l_0\}$	$\{pc = l_2\}$	$\{pc = l_2\}$	$\{pc = l_7\}$	$\{pc = l_7\}$
T_0	$\{pc = l_9\}$	$\{pc = l_9\}$	$\{pc = l_9\}$	$\{pc = l_9\}$	$\{pc = l_9\}$
I_0	**proof** $\Rightarrow$ broadcast $(\tau_{post}, A, \{pc = l_2\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{post}, B, \{pc = l_2\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{post}, C, \{pc = l_7\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{post}, D, \{pc = l_7\}_M)$	**violation** $\Rightarrow$ broadcast $(\tau_{vcond}, E, \{pc = l_7 \wedge x <= y\}_M)$
R_{start_1}	$\{pc = l_0\}$	$\{pc = l_2\}$	$\{pc = l_2\}$	$\{pc = l_7\}$	$\{pc = l_7\}$
T_1	T_0	$T_0 \cup \{pc = l_7 \wedge x \leq y\}$	$T_0 \cup \{pc = l_7 \wedge x \leq y\}$	T_0	T_0
I_1	*idle because noch change*	**violation** $\Rightarrow$ broadcast $(\tau_{vcond}, B, \{pc = l_2 \wedge y \neq 2 \wedge x \leq 99\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{post}, C, \{pc = l_7 \wedge x > y\}_M)$	*idle because no change*	*idle because no change*
R_{start_2}	$\{pc = l_0\}$	$\{pc = l_2\}$	$\{pc = l_2\}$	$\{pc = l_7 \wedge x > y\}$	$\{pc = l_7 \wedge x > y\}$
T_2	$T_1 \cup \{pc = l_2 \wedge y \neq 2 \wedge x \leq 99\}$	T_1	T_1	T_1	T_1
I_2	**proof** $\Rightarrow$ broadcast $(\tau_{post}, A, \{pc = l_2 \wedge y = 2 \wedge x < y\}_M)$	*idle because no change*	*idle because no change*	**proof** $\Rightarrow$ broadcast	**proof** $\Rightarrow$ broadcast
R_{start_3}	$\{pc = l_0\}$	$\{pc = l_2 \wedge y = 2 \wedge x < y\}$	$\{pc = l_2 \wedge y = 2 \wedge x < y\}$	$\{pc = l_7 \wedge x > y\}$	$\{pc = l_7 \wedge x > y\}$
T_3	T_2	T_2	T_2	T_2	T_2
I_3	*idle because no change*	**proof** $\Rightarrow$ broadcast $(\tau_{post}, B, \{pc = l_7 \wedge x > y\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{post}, C, \{pc = l_7 \wedge x > y\}_M)$	*idle because no change*	*idle because no change*
	Fixpoint reached $\Rightarrow$ Program safe				

In Line 12, the algorithm unpacks the postconditions (Algorithm 5) into abstract states, adding them to the initial precision π_0 for block b. It also unpacks the violation-condtion messages into target states (Algorithm 6) and adds them to the set T of target states, which also includes the original target states T_φ (Line 13).
After unpacking the messages, the algorithm runs the reachability analysis $\mathcal{A}$ on the current block b, starting from the unpacked postconditions and considering both the initial states and those in the waitlist. This analysis computes the set of reachable states R within the current block. This set is then used to extract the reachable abstract states E_R and the set of violated target states V. If V is not empty, the algorithm proceeds to broadcast the packed violation conditions (Algorithm 4) to the predecessor blocks of b (Line 18). In the case of V being empty, a stronger postcondition for the successor blocks of b is constructed, packed into a message (Algorithm 3) and broadcasted (Line 20).
The algorithm continues in this manner, iterating through the blocks and refining the analysis by sharing postconditions and violation conditions between the blocks until all reachable states and potential violations have been fully explored.

3.4.6 Example Run of DSS with Predicate Analysis

The following illustrates the workings of the DSS algorithm with Predicate Analysis on the program corresponding to the CFA shown in Figure 3.1. Initially the CFA is

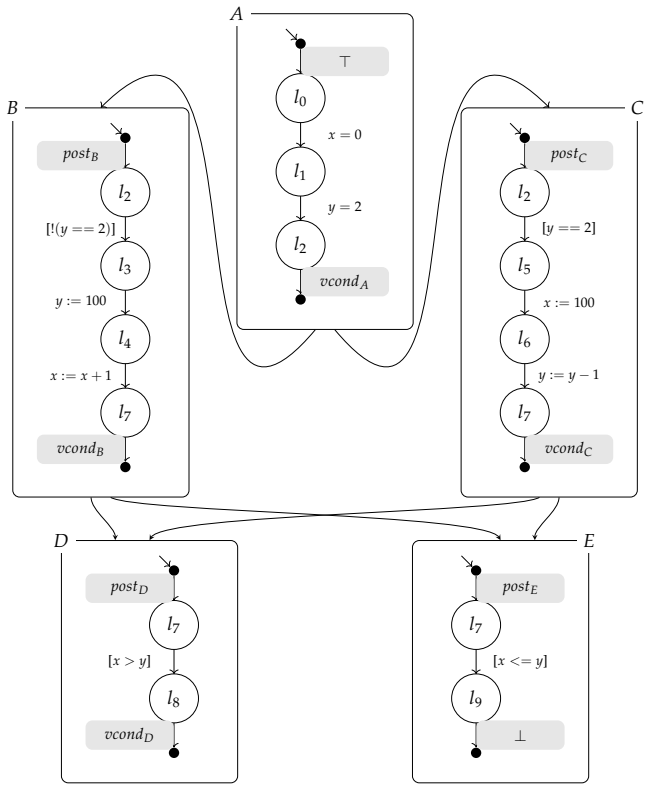


Figure 3.3: Linear Decomposition of Example Program

decomposed into blocks, where each block consists of a precondition, a block summary, which is computed using Craig Interpolation [11], and a violation condition. Each block, therefore, represents an independent verification task on which Predicate Analysis is iteratively run to further refine these components. For the sake of simplicity, we omit the possibility of integer overflows, as the implementation fully supports all required features. Additionally, we assume that, on every block, the same amount of time is consumed for running the analysis and message processing.

Decomposition

A valid decomposition of the CFA in Figure 3.1 is shown in Figure 3.3. The block graph is a result of the linear decomposition strategy, where each block consists of paths through the CFA where no branching occurs. Block A contains the initial part of the program where the variables x and y are initialized. Blocks B and C represent the two

branches after the initial block: one where $y \neq 2$ and another where $y = 2$. Block D and Block E represent the final states of the program which contain the assume statement. In block D the condition $x > y$ holds, which satisfies the program's specification. Block E however, contains the error state l_9, which when reached violates the specification of the program.

Analysis
Table 3.1 outlines the results of the DSS algorithm (Algorithm 2) with Predicate Analysis applied to the program. This table tracks the refinements of the preconditions, block summaries, and target states for each block node after every iteration. The preconditions are the initial abstract states for each block, while the postconditions are the abstract states after the analysis has been run. The block summaries are the results of the analysis, and the violation conditions are the target states that violate the specification. The table shows the results of the analysis after each iteration, with the initial abstract states for each block in the first row and the final results in the last row. The table demonstrates how the analysis refines the abstract states and block summaries through each iteration, leading to the final results.

Iteration 0. The analysis on each block start at the block entries with no initial information about the program state, therefore initial states R_{start_0} are equal to the start node of each block. The target states T_0 for analysis is the location l_9 of the failed assertion. Only the analysis on block E reaches a target state (violation) from which a violation condition is computed: If, at block entry l_7, condition $x \leq y$ holds, the program violates the specification. The violation condition is packed into a message, denoted with a subscript M, and broadcasted to all predecessor blocks. The analyses of all the other blocks reach no target states (proof) and therefore compute the trivial summary of location at their block exits and broadcast these summaries as their postconditions to all blocks.

Iteration 1. The predecessor blocks of E, B and C update their target states (T_1) with the information of the violation condition message from E. Running analysis on block B results in a violation due to the fact that x could be less than y. From that a violation condition is computed and communicated to its predecessor block A. The analysis on block C does not reach a target state and therefore computes a postcondition with the trivial summary of the block exit. The analysis on block C does not reach any target state and computes the summary $l_7 \wedge x > y$ which is communicated to all its successor blocks. All the other blocks, A, D and E are idle because they did not receive any new information.

Iteration 2. After receiving the post condition message from C, the successor blocks D and E update their initial reached sets R_{start_2}. Block A adds the information of the violation condition message given by B to its target state T_2. The analysis on block A shows that the target state cannot be reached, therefore a summary is computed and broadcasted to its successors. Blocks B and C did not receive any new information and are idle. The analyses on block D and E show that after updating their initial sets still no target states are reached.

Iteration 3. The blocks where the initial reached set and the target states did not change, namely A, D and E, are idle. The analyses on blocks B and C show that no target states are reached. Because the last broadcast of each block is a proof a fixed point is reached, the analysis terminates and the program is deemed as being safe.

4 Distributed Data-Flow Analysis in DSS

This chapter introduces the distributed data-flow analysis CPA and its integration into Distributed Summary Synthesis. The integration allows for an earlier top-down collaboration between the blocks by using the results of the data-flow analysis to strengthen the postconditions of the blocks and therefore the preconditions of the successor blocks.

4.1 Motivation

In the current implementation of DSS, Predicate Analysis is run on each block using CEGAR, meaning the abstraction is only refined in the case of a target state being reached or an error location being found. Because of this, blocks without an error state only provide trivial summaries to their successor blocks in the beginning of the analysis. Data-Flow Analysis, though less precise, can compute coarse quickly at summaries at the start of the analysis. The idea is to initially also run Data-Flow Analysis on each block to compute stronger postconditions. By communicating these summaries to their successor blocks, the precondition of the successor blocks can be strengthened.

4.2 Definition of Distributed Data-Flow Analysis

The distributed Data-Flow Analysis can be defined as a distributed CPA

$$\mathcal{D}_{\mathbb{I}} = (\mathbb{I}, \text{packPost}_{\mathbb{I}}, \text{unpackPost}_{\mathbb{I}}, \text{packVcond}_{\mathbb{I}}, \text{unpackVcond}_{\mathbb{I}})$$

with $\mathbb{I}$ being the invariants CPA defined in Section 3.2.4. The pack and unpack operators additionally define how the abstract states E of the invariants CPA $\mathbb{I}$ are packed into messages and unpacked from messages. The following paragraphs will discuss the pack and unpack operators of the distributed Data-Flow Analysis CPA.

The Pack Operator packPost$_{\mathbb{I}}$
The packPost operator of the distributed Data-Flow Analysis extracts all the invariants states E_{in} at the final location l_{exit} of the block b and computes the least upper bound. This information is packed into a postcondition message to be communicated between

the blocks. Specifically, the packPost operator works by summarizing all the invariants states which were computed at block exit into one message that can be passed along.

$$\text{packPost}_{\mathbb{I}} : 2^E \times B \to 2^M$$
$$\text{packPost}_{\mathbb{I}}(E_{in}, b) = \{(\tau_{post}, b, \{\sqcup E_{in}\}_M)\}$$

The Unpack Operator unpackPost$_{\mathbb{I}}$
The unpackPost operator of $\mathcal{D}_{\mathbb{I}}$ is responsible for unpacking a set of postcondition messages into invariants states. Every block which receives a post condition message from its predecessor blocks unpack the messages by joining all the invariants states and computing their least upper bound. If there is no such message from a predecessor block,, the unpackPost operator returns the top element $\top$ of the abstract domain of $\mathbb{I}$, namely a mapping function, in which every tracked variable is mapped to the interval $[-\infty, \infty]$.

$$\text{unpackPost}_{\mathbb{I}} : 2^M \times B \to 2^E$$
$$\text{invariantsStates}(M_{post}, b) := \begin{cases} \bigcup\{A \mid (\tau_{post}, b', A_M) \in M_{post}, b' \in \text{pred}(b)\}, & \text{if } \exists(\tau_{post}, b', A_M) \in M_{post}, b' \in \text{pred}(b) \\ \{\top\}, & \text{if no such } b' \text{ exists} \end{cases}$$
$$\text{unpackPost}_{\mathbb{I}}(M_{post}, b) = \{\bigsqcup \text{invariantsStates}(M_{post}, b)\}$$

The Pack Operator packVcond$_{\mathbb{I}}$
The packVcond operator is responsible for packing violation conditions into messages that can be communicated to the predecessor blocks. In the context of the distributed DFA, the packVcond operator is defined as always returning the empty set, because there is no bottom-up communication of violation conditions in the distributed DFA.

$$\text{packVcond}_{\mathbb{I}} : 2^E \times 2^E \times B \to 2^M$$
$$\text{packVcond}_{\mathbb{I}}(V, E_R, b) =$$

The Unpack Operator unpackVcond$_{\mathbb{I}}$
The unpackVcond operator also always returns the empty set, because within the distributed Data-Flow Analysis CPA, there are no violation condition messages passed during the analysis.

$$\text{unpackVcond}_{\mathbb{I}} : 2^M \times B \to 2^E$$
$$\text{unpackVcond}_{\mathbb{I}}(M_{vcond}, b) = \{\}$$

Table 4.1: Run of DSS with Predicate Analysis strengthened by Data-Flow Analysis

	A	B	C	D	E
R_{start_0}	$\{pc = l_0\}$	$\{pc = l_2\}$	$\{pc = l_2\}$	$\{pc = l_7\}$	$\{pc = l_7\}$
T_0	$\{pc = l_9\}$	$\{pc = l_9\}$	$\{pc = l_9\}$	$\{pc = l_9\}$	$\{pc = l_9\}$
$result_{df}$	$\{pc = l_2 \wedge x = 0 \wedge y = 2\}$	$\{pc = l_7 \wedge y = 100\}$	$\{pc = l_7 \wedge y = 1 \wedge x = 100\}$	$\{pc = l_8 \wedge x > y\}$	
I_0	**proof** $\Rightarrow$ broadcast $(\tau_{post}, A, \{pc = l_2 \wedge x = 0 \wedge y = 2\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{post}, B, \{pc = l_7 \wedge y = 100\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{post}, C, \{pc = l_7 \wedge y = 1 \wedge x = 100\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{post}, D, \{pc = l_7\}_M)$	**violation** $\Rightarrow$ broadcast $(\tau_{pcond}, E, \{pc = l_7 \wedge x <= y\}_M)$
R_{start_1}	$\{pc = l_0\}$	$\{pc = l_2 \wedge x = 0 \wedge y = 2\}$	$\{pc = l_2 \wedge x = 0 \wedge y = 2\}$	$\{pc = l_7 \wedge y = 100\}$ $\cup\{pc = l_7 \wedge y = 1 \wedge x = 100\}$	$\{pc = l_7 \wedge y = 100\}$ $\cup\{pc = l_7 \wedge y = 1 \wedge x = 100\}$
T_1	T_0	$T_0 \cup \{pc = l_7 \wedge x \leq y\}$	$T_0 \cup \{pc = l_7 \wedge x \leq y\}$	T_0	T_0
I_1	*idle because noch change*	**proof** $\Rightarrow$ broadcast $(\tau_{post}, B, \{false\}_M)$	**proof** $\Rightarrow$ broadcast $(\tau_{post}, C, \{pc = l_7 \wedge x > y\}_M)$	*idle because no change*	*idle because no change*
R_{start_2}	$\{pc = l_0\}$	$\{pc = l_7 \wedge x > y\}$	$\{pc = l_7 \wedge x > y\}$	$\{pc = l_7 \wedge x > y\}$	$\{pc = l_7 \wedge x > y\}$
T_2	T_1	T_1	T_1	T_1	T_1
I_2	*idle because no change*	*idle because no change*	*idle because no change*	**proof** $\Rightarrow$ broadcast	**proof** $\Rightarrow$ broadcast

Fixpoint reached $\Rightarrow$ Program safe

4.3 Adjustment of the DCPA Algorithm

The results of the data-flow analysis are integrated into the existing predicate analysis by adding the distributed invariants CPA $\mathcal{D}_{\mathbb{I}}$ to the composite CPA $\mathbb{C}$ of the current implementation of DSS. The transfer relation of the predicate CPA $\mathbb{P}$ strengthens their results with formula reporting states. Because the invariants state falls into that category, the strengthening happens automatically by forming a conjunction of the formulas of all formula reporting states at the block exit. To optimize the DCPA algorithm, the invariants state is excluded from the wrapped states in $\mathbb{C}$ when the predicate state at the block end is an abstraction state. This adjustment prevents unnecessary invocations of the strengthening function.

4.4 Example Run of DSS with Predicate Analysis strengthened by Data-Flow Analysis

To better illustrate the benefits of integrating Data-Flow Analysis into DSS, we will walk through an example run of DSS with Predicate Analysis, but this time strengthened by the results of DFA. The setup will be the same as in Section 3.4.6, but this time we will also run DFA on each block. The results of the DFA will be used to strengthen the postconditions of the blocks.

Analysis

Iteration 0. The start of the analysis mirrors the steps from Section 3.4.6. The initial

abstract states for each block correspond to the start node of each block. The target state T_0 for analysis remains the location l_9 where the assertion failure occurs. The analysis on block E reaches a target state (violation) from which a violation condition is computed, packed in a violation condition message and broadcasted to all its predecessor blocks. Simultaneously, DFA is run on each block. The DFA computes summaries, which are then be communicated to the successor blocks. For block A the DFA computes following condition that holds at block exit: $pc = l_2 \wedge x = 0 \wedge y = 2$. This information is used to strengthen the postcondition of block A and is communicated to its successor blocks B and C. Running DFA on block B provides blocks D and E with the information that at its block exit y has the value 100. On block C, DFA computes the following postcondition: $pc = l_7 \wedge y = 1 \wedge x = 100$. This information already indicates that the assertion cannot fail, when the program path leads through block C, because at its block exit x is definitely greater than y.

Iteration 1. In this iteration the effect of the DFA results on the analysis becomes clear. Each block starts with a stronger precondition because of the information provided by the predecessor blocks. The target states of blocks B and C are updated with the communicated violation condition of E just like in the example run without DFA. Now when running predicate analysis on block B no target states in T_1 are reached. Due to the contradiction of $y = 2$ and $y \neq 2$ the analysis computes the summary *false*. With the updated initial reached set R_{start_1} the analysis on block C also does not reach any target states. The summary of block C is computed as $pc = l_7 \wedge x > y$. The analysis on block D and E are idle because they haven't received any new information.

Iteration 2. Only D and E received new information from their predecessor blocks. The analysis shows that this does not change the fact that no target state is reached. All the other blocks are idle because their initial reached set and target states haven't change. The analysis therefore reaches a fixpoint because the last broadcast of each worker does not contain a violation condition. After only two iterations it is shown that the program is safe.

5 Implementation

In this chapter we will discuss the details of the implementation of the Distributed Data-Flow Analysis CPA, which is implemented in CPAchecker [4], a configurable software verification framework written in Java.

5.1 Distributed Data-Flow Analysis

5.1.1 Serialization of Invariants State

In this section, we describe how the invariants state is serialized to communicate the results of the DFA between the blocks within the block graph. The serialized state contains a boolean formula, a variable types map, and an abstraction strategy that allows the receiving successor block to accurately reconstruct the invariants state passed through a message.

Visitor Pattern for Serialization
The visitor pattern in java is used to add new behaviour to existing object structures without modifying the structures. The visitor pattern is implemented by defining a visitor interface which declares a visit method for each type of object in the object structure. The object structure is then defined by a class which implements the accept method. The accept method is used to call the visit method of the visitor. The visitor pattern is used to traverse the object structure and to call the visit method of the visitor[.]Visitor

Serialization of Boolean Formula
The boolean formula stored in the invariants state captures the relationships between and the possible values of the tracked variables at the end of a block. For serialization, the `SerializeBooleanFormulaVisitor` class (Figure 5.1) is used to traverse the boolean formula, such as Equal and LessThan, and translates each into a string representation defined in the `Operation` enum. For instance, a logical "and" operation is transformed into "&&.la" and a logical "or" operation is transformed into "||.lo". This approach allows for easy adjustments to the string representation of each operation and operand. After traversing the boolean formula, the serialized formula is stored in

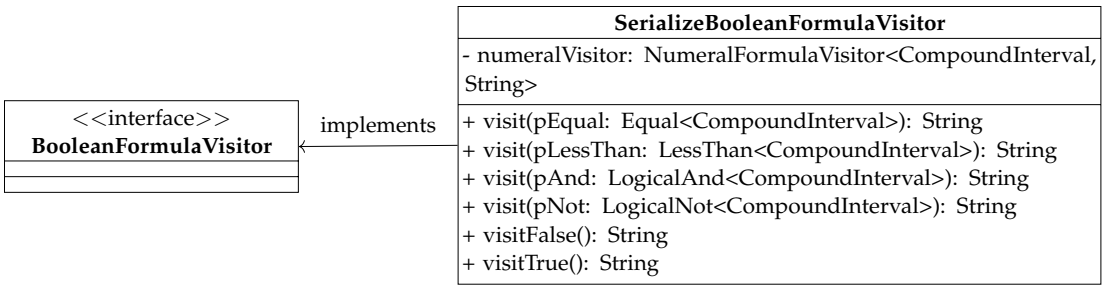


Figure 5.1: UML diagram of implementation of the boolean formula visitor

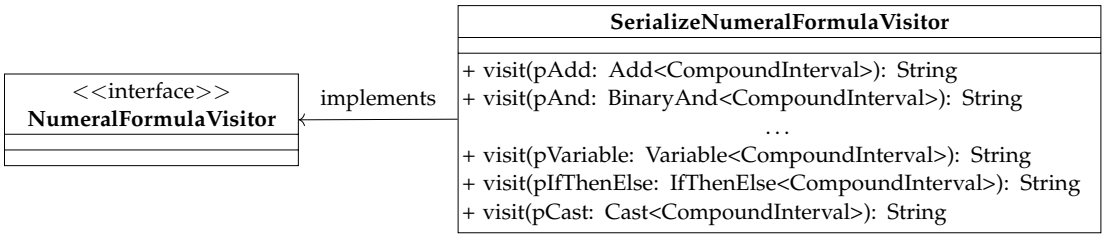


Figure 5.2: UML diagram of implementation of the numeral formula visitor

the `BlockSummaryMessagePayload` object.

Serialization of Variable Types Map
The variable types map associates the memory locations of the tracked variables with their type. The serialization of the variable types map is necessary to reconstruct the invariants state of the block. For this, each key-value pair is brought into a string format where ".ti" separates the memory location from its type. All these pairs are then joined by "&&". To serialize the memory location the qualified name, which includes the variable name and the function in which it is located, is used. Similiar to the boolean formula, the type is serialized by traversing through the CType with the `SerializeCTypeVisitor` (Figure 5.3). The serialized map is added as an own entry in the `BlockSummaryMessagePayload`.

Serialization of Abstraction Strategy
The abstraction strategy of the invariants state determines how and when the analysis transitions between abstract states. There are predefined strategies, such as ALWAYS, ENTERING_EDGES and NEVER, in the `AbstractionStrategiesFactories` enum. For the serialization we rely on the method which returns the string representation of the used abstraction strategy. Again, this is stored in the `BlockSummaryMessagePayload` as a separate entry.

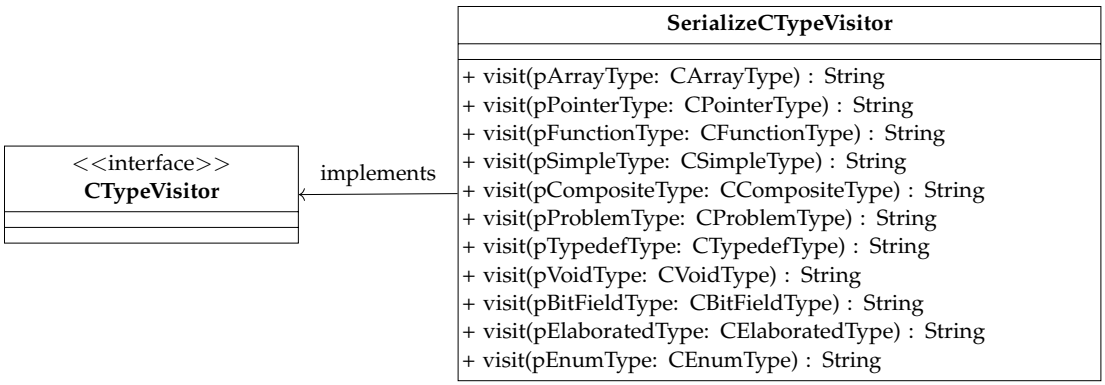


Figure 5.3: UML diagram of implementation of the cType visitor

After adding each component, which is necessary to reconstruct the invariants state, to the `BlockSummaryMessagePayload`, the payload is built and can be sent to the successor blocks.

5.1.2 Deserialization of Invariants State

The deserialization process is responsible for reconstructing the serialized invariants state from the Block Summary Message Payload received by each block. The boolean formula, variable types map and abstraction strategy are extracted from the payload and used to restore the serialized invariants state.

Deserialization of Boolean Formula
The `StringToBooleanFormulaParser` interprets the serialized boolean formula string and reconstructs the logical structure. Using a bracket stack, the parser keeps track of the nested structure of the formula and applies the `Operation` enum symbols to identify each operation and operand. For example, "&&.la" would be recognized as a logical "and" between two conditions and therefore restore the original structure of the boolean formula.

Deserialization of Variable Types Map
The variable types map is reconstructed by splitting the serialized string at each "&&" to separate entries, and then again at ".ti" to extract the qualified name and type of each variable. Each qualified name is then converted back to a memory location and the type is restored by using the `CTypeParser` class, which works similar to the `StringToBooleanFormulaParser`.

```
--predicateAnalysis-block
--option distributedSummaries.worker.forwardConfiguration=config/
   distributed-block-summaries/predicateAnalysis-dataFlow-block-
   forward.properties
<path to program>
```

Figure 5.4: Configuration to run the distributed predicate analysis with DFA

Deserialization of Abstraction Strategy
The abstraction strategy is restored by using the `valueOf` method which is defined in the `AbstractionStrategiesFactories` enum. The string representation of the abstraction strategy is used to create the same abstraction strategy as the one used in the serialized invariants state.

5.2 Limitations

This described implementation has some limitations, which might be impacting the overall performance of the approach. First, the invariant state is computed exclusively at the end of each block and is used to strengthen the predicate state only when it is an abstraction state. This approach results in some inefficiencies, as it necessitates recomputation at the end of each block rather than at more strategically defined stages of the analysis.
Furthermore, while the data-flow analysis (DFA) is intended to run primarily at the beginning of the analysis to provide a robust postcondition for each block, it currently executes in each iteration. This setup is particularly complex for loop structures, where blocks reference themselves, complicating the restriction of DFA to initial iterations. Restricting DFA to the beginning of the analysis, especially in loops, would require additional refinement to avoid added computational overhead.
Finally, there are specific limitations in the implementation of the CType visitor. For CompositeType, the serialization process currently excludes its members, leading to incomplete representations when complex data types are used. Similarly, for elaborated types, complex types are not serialized. During development, cyclic dependencies—such as a composite type containing a pointer referencing one of its own types—caused infinite recursion in the visitor class. While this issue could be mitigated by introducing a mapping mechanism that pairs each type name with its serialized CType, such a solution would add significant overhead for a relatively minor gain, and thus was not included in this implementation.

5.3 Configuration

In CPAtextscchecker, a configuration already exists for running the standard predicate analysis within DSS. To integrate DFA alongside predicate analysis, we created a custom forward configuration that extends the existing setup. In this new configuration, the invariants CPA is added to the composite CPA to enable DFA. Additionally, other options are set to manage how the forward analysis is configured, such as when strengthening should be applied or how abstract states should be merged. These configurations allow DFA to run concurrently with predicate analysis. To execute the distributed predicate analysis with DFA, we simply have to set the new forward configuration in the command line as shown in Figure 5.4. Additionally, users can specify a decomposition strategy other than the default merge decomposition by setting the option `distributedSummaries.worker.decompositionStrategy` to the desired strategy.

6 Evaluation

6.1 Experimental Setup

For the evaluation, we conduct a series of experiments to compare the performance of DSS with and without data-flow analysis. The following hardware configuration was used for the experiments:

- CPU Model: Intel Xeon E3-1230 v5 @ 3.40 GHz
- Memory Limit: 15 GB
- CPU Cores: 8

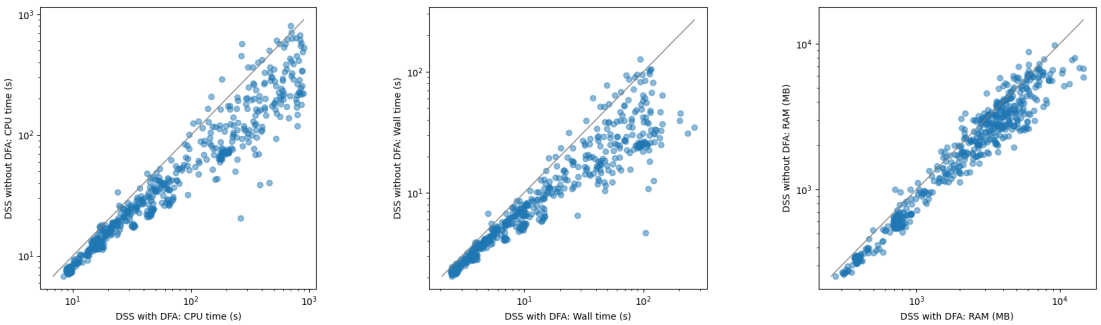
Each run was given a time limit of 900 seconds, with a hard time limit of 1000 seconds. The maximum heap size allocated for each run was 13 GB.

We ran a total of 3402 tasks using the following test-sets of the SV-COMP 24 [2] benchmarks[1]:

- SoftwareSystems-AWS-C-Common-ReachSafety
- SoftwareSystems-BusyBox-ReachSafety
- SoftwareSystems-coreutils-ReachSafety
- SoftwareSystems-DeviceDriversLinux64-ReachSafety
- SoftwareSystems-DeviceDriversLinux64Large-ReachSafety
- SoftwareSystems-uthash-ReachSafety
- ReachSafety-ProductLines

All experiments are conducted using revision 8c6ce7f1 of CPAchecker[2]. The complete set of artifacts, including experimental data and configurations, can be found on Zenodo[3].

[1] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/commit/5c2dfcdb832719de3145e5ed3ba88550cd076eef
[2] https://gitlab.com/sosy-lab/software/cpachecker/-/tree/8c6ce7f1d85b88cd60786bab6902b5e311ed3738
[3] https://doi.org/10.5281/zenodo.13959611



(a) CPU time (b) Wall time (c) Memory usage

Figure 6.1: Comparison of resource consumption between DSS with and without DFA.

6.2 Research Questions

The integration of DFA into DSS presents opportunities for improved resource efficiency, enhanced verification coverage, and reduced communication overhead. However, the additional computational requirements, message exchanges, and serialization and deserialization overheads introduced by integrating DFA may also present challenges. To explore both the benefits and limitations of integrating DFA into DSS, we evaluate the discussed approach by answering following research questions:

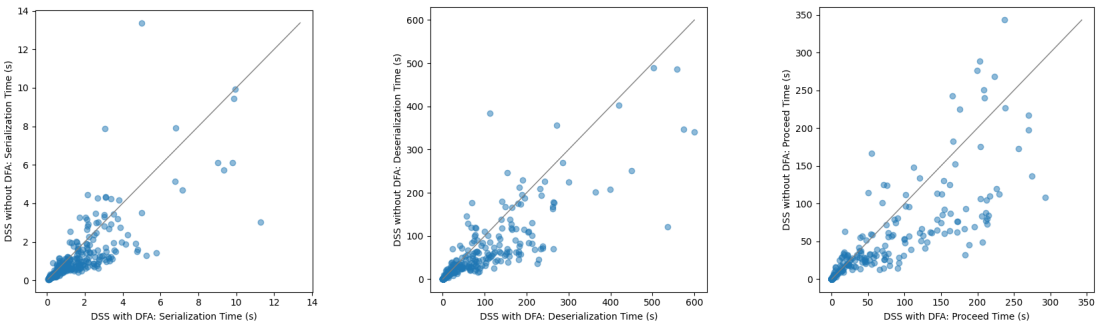
RQ 1: Change of Resource Consumption. How does the integration of Data-Flow Analysis into the Distributed Summary Synthesis (DSS) framework impact resource consumption, particularly in terms of CPU time, wall time, and memory usage?

RQ 2: Change of Verification Coverage. Are there programs that benefit uniquely from either the standard predicate analysis or the predicate analysis strengthened by Data-Flow Analysis? What characteristics define these programs?

RQ 3: Change of Message Overhead. How does the integration of Data-Flow Analysis affect the communication overhead in DSS?

6.3 Experimental Results

In this section, we analyse the experimental results we obtained from running DSS with and without DFA on the tasks mentioned above. The aim of running these experiments is to evaluate how the integration of DFA impacts resource consumption, verification



(a) Serialization Time (b) Deserialization Time (c) Proceed Time

Figure 6.2: Comparison of Serialization and Deserialization time.

coverage, and message overhead in DSS. By plotting the results in scatter plots, we compared both configurations across multiple metrics, with each point representing a program that was verified correctly by both configurations. Each scatter plot shows the results of DSS with DFA on the x-axis and without DFA on the y-axis. The diagonal line represents the case where both configurations perform equally well.

6.3.1 Change of Resource Consumption

The scatter plots in Figure 6.1 illustrate the impact of DFA on CPU time, wall time, and memory usage. The following passages examine each of these aspects in detail.

CPU Time
The scatter plot for CPU time in Figure 6.1a shows that DSS with DFA generally requires more CPU time than without DFA. This is indicated by the majority of points lying below the diagonal. This becomes especially evident for programs which already required a significant amount of CPU time without DFA. The increase in CPU time is likely due to the additional cost of serializing and deserializing the invariants state. The plots in Figure 6.2a and Figure 6.2b show how much more time is spent on serialization and deserialization with DFA. Moreover, the proceed operator, which determines the continuation of analysis by checking the termination condition, takes up significantly more time with DFA. This is shown in Figure 6.2c.
While there are a few instances where DSS with DFA completes with less CPU time, on average DSS with DFA requires more CPU time than DSS without DFA. This highlights a clear short coming: while DFA provides the analysis with more information, the introduced overhead is too big.

Wall Time
The scatter plot for wall time in Figure 6.1b reveals a similar pattern to that observed with CPU time. Wall time measures the actual elapsed time to verify a program from start to finish. As with CPU time, DSS with DFA generally results in longer wall times, indicated by the clustering of points below the diagonal. This increase is very likely due to the additional overhead introduced by DFA, including serialization and deserialization of the invariants state and the costly proceed operator. The difference becomes more pronounced for tasks that already required significant wall time without DFA, suggesting that, for complex programs that are already time-intensive, DFA's added information comes with a time cost that is no longer beneficial for the overall program analysis.

Memory Usage
The scatter plot for memory usage in Figure 6.1c shows that DSS with DFA generally consumes more memory than the configuration without DFA, as indicated by the clustering of points below the diagonal. The integration of DFA introduces a consistent, moderate increase in memory usage across all programs, due to additional messages generated early in the analysis. These messages that hold the serialized invariants state, holding the boolean formula, variable types map and abstraction strategy, add to the memory demand to each program. However, the increase in memory usage is relatively minor compared to the impact on CPU and wall time.

Summary
The integration of DFA into DSS generally increases resource consumption. Both CPU time and wall time show significant increases, particularly for more complex programs, due to the overhead from DFA computations and communication. While memory usage also increases with DFA, this impact is relatively minor compared to CPU and wall time. Overall, while DFA provides the analysis with more information, it introduces noticeable resource costs.

6.3.2 Change of Verification Coverage

To assess the impact the integration of DFA might have on the verification coverage, we compared the number of programs successfully verified with and without DFA. Out of 6402 tasks, the configuration without DFA was able to verify 887 programs correctly, while with DFA, the number was lower at 672 correctly verified tasks (Table 6.1). This difference indicates that the added computational overhead through the serialization and deserialization of the invariants state and the additional messages exchanged between the workers reduces the number of programs that can be verified within

Table 6.1: Results of verified programs with and without DFA

Result	DSS with DFA	DSS without DFA
correct true	600	808
correct false	72	79
incorrect true	0	14
incorrect false	1	1

Table 6.2: Sample of programs which were able to be verified only via Predicate Analysis using Data-Flow Analysis

Task	Status_{new}	CPU Time_{new} (s)	Wall Time_{new} (s)	Status_{old}
32_1_cilled_ok_nondet_linux-3.4-...	true	738	135	TIMEOUT
32_1_cilled_ok_nondet_linux-3.4-...	true	745	99.7	TIMEOUT
minepump_spec2_product03.cil.yml	true	803.81	122.88	TIMEOUT
minepump_spec2_product04.cil.yml	true	694.88	107.68	TIMEOUT
minepump_spec2_product11.cil.yml	true	760.97	117.19	TIMEOUT

resource limits.
In terms of soundness, Table 6.1 shows that the integration of DFA does not introduce any new incorrect verification results. Both configurations have a single incorrect false verification verdict. This consistency confirms that the integration of DFA does not compromise the soundness of the analysis.
Further analysis revealed that the benefits and limitations of DFA integration are primarily observed within the mine pump system, a subset of the product lines category. This system includes two sets of programs where DFA proved advantageous, namely programs that could only be verified with DFA and would otherwise timeout (Table 6.2), and programs that were verified faster with DFA than without it (Table 6.3). This suggests that DFA's initial coarse summaries help prune infeasible paths early in the verification process. By providing concrete interval summaries at the block exit, DFA can reveal certain variable constraints that make some blocks logically unreachable. As a result, Predicate Analysis can skip these blocks or paths without fully exploring them, reducing the overall required computation and refinement steps. This early path pruning is especially beneficial in programs with simple branching conditions, where eliminating a few paths can significantly reduce verification complexity.
However, the mine pump category also included nearly a hundred programs that could be verfied without DFA but led to timeouts when DFA was integrated. A plausible explanation is that DFA's precise interval summaries make it challenging for Predicate

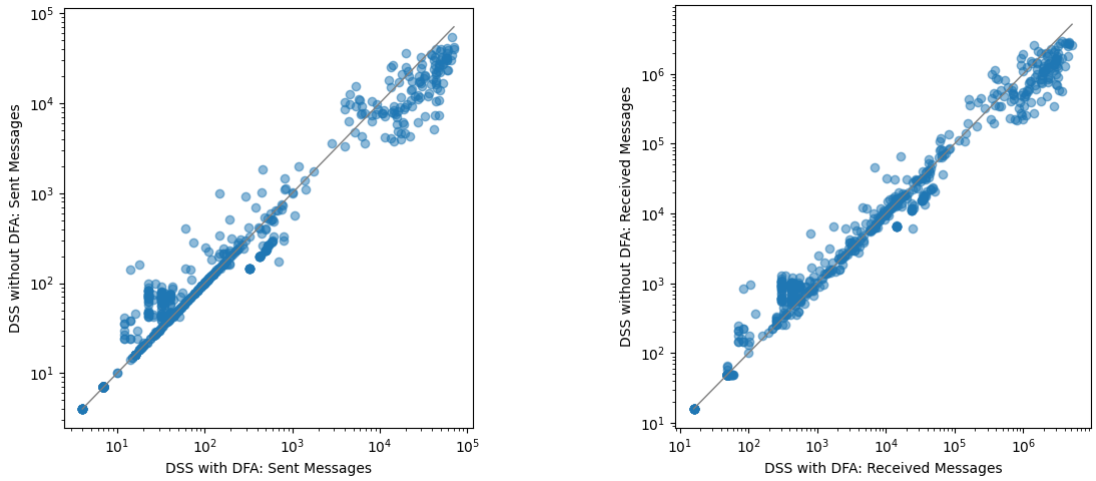
Table 6.3: Sample of programs which were verifed quicker (shorter wall time) using Predicate Analysis strengthened by Data-Flow Analysis

Task	Wall Time$_{new}$ (s)	Wall Time$_{old}$ (s)
minepump_spec2_product05.cil.yml	94.31	126.57
minepump_spec2_product06.cil.yml	72.07	93.42
minepump_spec3_product20.cil.yml	37.70	60.91
minepump_spec3_product52.cil.yml	23.71	31.23
minepump_spec3_product59.cil.yml	43.67	51.77

Analysis to find suitable over-approximations. The results of the DFA often represent concrete ranges that are difficult for Predicate Analysis to abstract effectively, leading to repeated refinement steps. These additional refinement steps result in higher computational cost and more messages being exchanged, which ultimately can lead to more timeouts.
These results indicate that while DFA improves efficiency in certain cases, its integration can introduce excessive overhead in more complex scenarios where generalizing detailed interval data can be counterproductive.

6.3.3 Change of Message Overhead

To examine how DFA changes the message overhead, we compared the number of sent and received messages for both configurations, as shown in Figure 6.3. The scatter plot reveals distinct patterns in the behaviour of message exchange depending on the complexity of the programs.
For simpler programs, where fewer messages are already exchanged, DFA further reduces the number of sent and received messages. This suggests that DFA allows these programs to reach verification conclusions more efficiently, requiring less communication between blocks. In the middle range of program complexity, there is little change in the number of messages exchanged with and without DFA. However, for more complex programs, those with numerous blocks or many loops, the integration of DFA leads to a noticeable increase in message overhead.
These observations suggest that while DFA can reduce message overhead for simpler programs, it may introduce additional overhead for more complex programs. One potential explanation could be that DFA, by tracking and providing concrete values in the form of intervals, might lead to more detailed states. This added detail could challenge Predicate Analysis when it attempts to maintain a suitable level of abstraction.



(a) Sent Messages Comparison

(b) Received Messages Comparison

Figure 6.3: Comparison of Sent and Received Messages.

6.3.4 Threats to Validity

This section addresses potential limitations and sources of bias in the experimental results, focusing on external and internal validity.

External Validity
Our evaluation uses a subset of tasks from the SV-COMP 2024 [2] benchmark set. While this is a widely accepted collection, it may not fully represent the range of real-world programs, which could limit the generalizability of our findings. Additionally, since our implementation relies on CPAchecker's predicate analysis, results may vary if applied to different verification tools or configurations.

Internal Validity
Several factors could influence the accuracy of our results. Our implementation may contain undetected bugs and still has limitations (Section 5.2), which can lead to unexpected results. Additionally, measuring CPU time per thread could lead to minor timing inaccuracies because the measurement happens at the CPU core, which is difficult and can therefore be inaccurate. Finally, the non-deterministic scheduling of message processing may introduce some variability in analysis duration because the order in which messages are processed can have an impact on the duration of the analysis.

7 Conclusion

In this thesis, we extended Data-Flow Analysis to operate as a distributed analysis within the Distributed Summary Synthesis framework. The primary contributions of this work comprise the formal definition of distributed DFA and its implementation in CPAchecker. This in turn required developing a serialization and deserialization process for the invariant states. Additionally, we introduced a new configuration that integrates DFA into the existing DSS setup. By running DFA in parallel with Predicate Analysis, blocks can generate quick, coarse summaries for their successors at the start of the analysis, potentially enhancing the efficiency of DSS.
DFA was shown to be beneficial only in specific cases, where its early summaries might have helped streamline verification by pruning infeasible paths. This allowed some programs to be verified quicker with DFA than without DFA, and, in other cases, enabled successful verification where Predicate Analysis alone would have timed out. However, in more complex programs, DFA's detailed intervals seemed to require more refinement steps and memory, which likely contributed to increased timeouts. Despite these variations, the integration of DFA did not compromise soundness, as both configurations produced consistent verification outcomes.
Furthermore, deficiencies of the current implementation, such as incomplete support for certain data types, may have influenced these results. Addressing these limitations could improve verification coverage and performance.
Overall, the findings of this thesis suggest that although DFA has potential benefits for verification in some contexts, its integration in DSS requires careful balancing of the level of provided information and resource consumption. Future work could explore ways to adjust DFA's detail level to reduce unnecessary overhead or apply DFA selectively in parts of the program where it is most useful.

Abbreviations

ABE Adjustable Block-Encoding

SBE Single-Block Encoding

LBE Large-Block Encoding

BAM Block-abstraction Memoization

DFA Data-flow Analysis

CPA Configurable Program Analysis

CFA Control Flow Automaton

CEGAR Counterexample-Guided Abstraction Refinement

CPA+ CPA with dynamic precision adjustment

DSS Distributed Summary Synthesis

List of Algorithms

List of Figures

List of Tables

Bibliography

[1] D. Beyer, M. E. Keremoglu, and P. Wendler. "Predicate abstraction with adjustable-block encoding." In: *Formal Methods in Computer Aided Design*, pp. 189–197.

[2] D. Beyer. "State of the Art in Software Verification and Witness Validation: SV-COMP 2024." In: *Proceedings of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024, Luxembourg, Luxembourg, April 6-11), part 3*. Ed. by B. Finkbeiner and L. Kovács. LNCS 14572. Springer, 2024, pp. 299–329. DOI: 10.1007/978-3-031-57256-2_15.

[3] D. Beyer, T. Henzinger, and G. Theoduloz. *Program Analysis with Dynamic Precision Adjustment*. 2008, pp. 29–38. ISBN: 978-1-4244-2187-9. DOI: 10.1109/ASE.2008.13.

[4] D. Beyer, T. A. Henzinger, and G. Théoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis." In: *Computer Aided Verification*. Ed. by W. Damm and H. Hermanns. Springer Berlin Heidelberg, pp. 504–518. ISBN: 978-3-540-73368-3.

[5] D. Beyer, M. Kettl, and T. Lemberger. "Decomposing Software Verification using Distributed Summary Synthesis." In: *Proc. ACM Softw. Eng.* 1.FSE (2024). DOI: 10.1145/3660766.

[6] A. Cimatti. "Software Model Checking via Large-Block Encoding." In: (2009). DOI: 978-1-4244-4966-8.

[7] J. Fischer, R. Jhala, and R. Majumdar. *Joining dataflow with predicates*. Conference Paper. 2005. DOI: 10.1145/1081706.1081742.

[8] C. Hewitt. "Actor Model of Computation: Scalable Robust Information Systems." In: arXiv (2010).

[9] H. Jain, F. Ivančić, A. Gupta, I. Shlyakhter, and C. Wang. "Using Statically Computed Invariants Inside the Predicate Abstraction and Refinement Loop." In: *Computer Aided Verification*. Ed. by T. Ball and R. B. Jones. Springer Berlin Heidelberg, pp. 137–151. ISBN: 978-3-540-37411-4.

[10] J. C. Knight. "Safety critical systems: challenges and directions." In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pp. 547–550.

[11] K. L. McMillan. "Interpolation and SAT-Based Model Checking." In: *Computer Aided Verification*. Ed. by W. A. Hunt and F. Somenzi. Springer Berlin Heidelberg, pp. 1–13. ISBN: 978-3-540-45069-6.

[12] D. Wonisch and H. Wehrheim. "Predicate Analysis with Block-Abstraction Memoization." In: *Formal Methods and Software Engineering*. Ed. by T. Aoki and K. Taguchi. Springer Berlin Heidelberg, pp. 332–347. ISBN: 978-3-642-34281-3.