Bachelor's Thesis in the Field of Computer Science

Implementation and Evaluation of Backwards Analyses in the Software-Verification Framework CPAchecker

Written by
Andrea Kreppel
born on the 5th of September, 1992
in Passau, Germany

for the Bachelor's Degree in Computer Science at the
Ludwig-Maximilians-Universität München.

Handed in at the Examination Board:	**17th of September, 2019**
Date of the Oral Examination:	**18th of September, 2019**
Supervisor:	**Prof. Dr. Dirk Beyer**
Advisor:	**Karlheinz Friedberger**

Declaration of Authenticity

I hereby declare that this thesis arose under the supervision of Prof. Dr. Dirk Beyer and is the result of my own work and investigation, except where otherwise stated. Any literature, data or work done by others is given due acknowledgment in this thesis by citation and listing in the reference section.

This work was not previously presented to another examination board and has not been published. The advisor of this thesis is fully entitled to use this or parts of this work for further publications.

Munich, September 11, 2019

Andrea Kreppel

Abstract

Software model checking has become a powerful technique in program verification. In spite of its success, it still suffers from the 'state explosion problem'. Although there have been many improvements concerning this matter, there are still many verification tasks remaining which are not yet manageable. Some of these challenging programs might be feasible if the analysis takes a different traversal route through the program avoiding the difficult structure. A way of altering the traversal route of an analysis is to change the analysis direction.

In this work, the analysis direction is compared for two symbolic model checking approaches, a BDD based program analysis and a predicate based analysis, on a subset of the SV-COMP 19 repository. The comparison reveals that the analyses in the traditional forwards direction outperform in most cases the respecting analysis in backwards direction.

Acknowledgments

First of all, I would like to thank Professor Beyer for the opportunity to write this thesis in his group at the chair of Software and Computational Systems Lab of the University of Munich (LMU).

Special thanks goes to my advisor Karlheinz Friedberger for the excellent support. Whenever I needed help or had a question I could rely on a good advice coming in almost instantly via email.

Furthermore, I would like to thank Andreas Baumann for mental support as well as covering my back so that I could concentrate on the thesis. Moreover, I thank my parents for letting me do whatever I want but give me their unquestioning support. Last but not least, I want to thank Ingrid and Werner Baumann for providing comfortable surroundings for writing this thesis.

Contents

1 Introduction

Nowadays, software is present in nearly all areas of everyday life. Consequently, the efficient and bug free software development becomes increasingly important. Not only that erroneous software often involves a huge economic loss but can also be dangerous if safety-critical systems are affected. Model checking has become a powerful approach in software verification that can find a large number of bugs, thus representing a promising alternative to software testing [1]. One of the main challenges in model checking is the 'state explosion problem' [2]. This arises when the investigated system has many components which make transitions in parallel. An enormous advance concerning this matter was the introduction of binary decision diagrams (BDDs) for the representation of transition relations [3]. With this, a larger number of states could be handled [4], [5]. Another improvement for tackling the 'state explosion problem' is the counterexample-guided abstraction refinement (CEGAR) [6], [7]. This technique reduces the state space by starting the analysis with a coarse model of the system and iteratively refining the model until it is detailed enough for a reliable verification result. In spite of these advances, there are still many verification tasks that exceed the available capacities.

A different approach for coping with this circumstance is to avoid the challenging structure of not feasible programs. This can be attempted by investigating the program starting at the exit locations and not from the beginning as it is commonly done. Backwards traversal has already been applied in hardware model checking [8] as well as in symbolic software model checking based on BDDs [9]. The power of the combination of forwards and backwards traversal has been showed for hardware model checking [10], software verification with BDDs [11]–[13], abstract interpretation [14] or using Craig interpolants [15], [16]. As a program analysis in backwards direction takes a different run through a program, it is possible that it manages to verify programs that the corresponding forwards analysis is not able to tackle.

The aim of this work is to compare the backwards program analysis approach to the forwards program analysis. For this reason two symbolic model checking approaches are implemented in CPAchecker [17] for the use in backwards analysis direction. The first one is the BDD based program analysis [18] and the second one is the predicate based program analysis [19], [20] that is compatible for the use in the CEGAR algorithm. These two approaches are compared in forwards and backwards analysis direction using bread-first and depth-first traversal on a selection of tasks of the SV-COMP 19 repository [21].

2 Theoretical Background

2.1 Software Model Checking

Software model checking is an approach to prove the correctness of a program according to a given specification. One approach for this is reachability analysis, in which it is investigated if an execution path for a program exists that reaches a specified program location. This is often done by construction of an abstract reachability graph (ARG) [22], [23]. An ARG is build up by traversing the control flow automata (CFA) of the program and sequentially calculating the successor state(s) using the information of the edges of the CFA [19].

Example 1 Example program adopted from Ref. [19].

```
1: int main() {
2:   int i = 0;
3:   while (i < 2) {
4:     i++;
5:   }
6:   if (i != 2) {
7:     ERROR: return 1;
8:   }
9: }
```

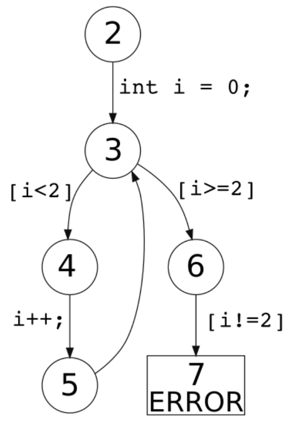


Figure 1: Control flow automaton of the example program shown in Example 1. The nodes represent program locations with the respecting line number as label. The edges represent program statements. Assumptions like `assume(p)` are denoted as `[p]`.

The nodes of an ARG are abstract states which contain more domain specific data like control-flow location, call stack information, and a path formula which represent the data state [24]. The edges of the ARG represent the program operations corresponding to the edge that was followed in the CFA. For illustration of the construction of an ARG using single-block encoding (SBE) [25], the example program shown in Example 1 and its corresponding CFA (Figure 1) are considered. The reachability analysis starts normally at the entry of the `main` function (root of the CFA), so the first abstract state that is represented by the first node of the ARG (Figure 2) contains the program location before execution of line 2.

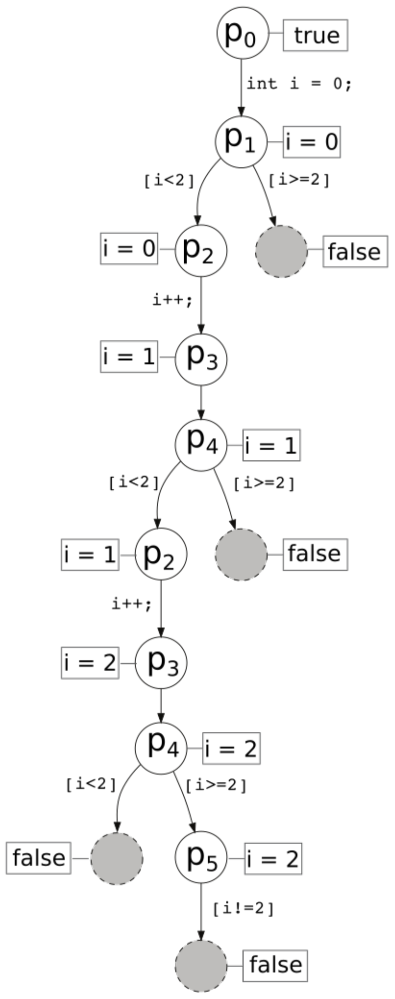


Figure 2: Abstract reachability graph for the example program shown in Example 1.The abstraction of the path formula of the respecting abstract state (node) is attached in a box. The edges contain as the CFA (Figure 1) the program statements.

The successor abstract state is obtained by interpreting the next program operation that is succeeding the current location state (label of the outgoing edge of the corresponding CFA node). The resulting path formula of the computed abstract state corresponds to the strongest postcondition of the path formula of the predecessor and the interpreted program operation. In this way the CFA is enrolled into an ARG by successively going along the CFA edges and appending the information to the path formula until either the additional information gives a contradiction together with the path formula of the predecessor state resulting in a path formula '*false*' or the location of interest l_{ERR} (here an error label) is reached without conflict in the path formula. The latter case proves that l_{ERR} can be reached by the execution path documented in the ARG from the root to this leaf. A path formula of '*false*', as in the first case, for a given abstract state reveals that the contained program location under the given conditions is not reachable. So a complete ARG has either the a path formula '*false*' in all of its leafs as the ARG of the example program (Figure 2), which implies that the l_{ERR} can not be reached, or it has a leaf with a path formula without contradiction and l_{ERR} as location state proving that this location is reachable. Despite from the SBE also large-block (LBE) or adjustable-block encoding (ABE) can be applied. The difference here from be SBE is that the path formula is not recomputed after interpreting every single program operation. In the LBE the CFA is transformed into a 'summarized' CFA where each edge contains a block of the same predefined number of program operations [24]. In ABE indeed every single program operation is interpreted but the newly gained information is not integrated into the complete information of the current path (abstraction formula) but stored in a second formula (path formula) and in intervals of adjustable length these two formulas are combined and evaluated [19]. Thus, ABE also summarizes multiple CFA edges and represents a generalization of LBE by summarizing a variable number of operations in each block. These two approaches have the advantage that the costly evaluation of the complete path formula does not take place for every single program operation.

2.1.1 Configurable Program Analysis

For a reliable comparison of different model checking and program analysis approaches it is essential to express them in the same formal setting. This is enabled by the CPAchecker framework [17] that provides an interface for the definition of program analyses via the configurable program analysis (CPA) formalism. For the CPA formalism, it is assumed that the program analysis operates on a program represented by an CFA $A = (L, l_{INIT}, G)$ which consists of a set of program locations L, an initial location l_{INIT}, and a set $G \subseteq (L \times Ops \times L)$ of edges representing the program operations [23].

A CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$ consists of an abstract domain D, a transfer relation $\rightsquigarrow$, a operator 'merge', and a termination check 'stop'. The abstract domain $D = (C, \mathcal{E}, [[\cdot]])$ comprises a semilattice $\mathcal{E} = (E, \sqsubseteq)$ over a set of partially ordered abstract-domain elements. The concretization function $[[\cdot]]$ assigns to each abstract state $e \in E$ the set of concrete states C e represents. Furthermore the CPA has a transfer relation $\rightsquigarrow \subseteq E \times G \times E$ which assigns to each abstract state e all possible successor states e' while each transfer can be mapped onto a control-flow edge $g \in G$: $e \overset{g}{\rightsquigarrow} e'$. The following theoretical description is limited to programs with only assume or assignment statements. The operator 'merge': $E \times E \rightarrow E$ defines if and how the information of two abstract states are combined and the termination check 'stop': $E \times 2^E \rightarrow \mathbb{B}$ probes if a given abstract state is already covered by the remaining set of abstract states.

The CPA interface makes use of the composite pattern enabling the combination of several CPAs using a 'Composite CPA' where the abstract states are tuples of the abstract states of each component CPA and the operators delegate to the respecting operator of the component CPAs [26]. For most analysis purposes the Composite CPA contains the Location CPA $\mathbb{L}$ which traces the reachability of CFA locations and an ARG CPA $\mathbb{A}$ which enables the construction of an ARG by memorizing the predecessor-successor relationships between abstract states [27].

The core algorithm of the CPA formalism (Algorithm 1) is a reachability analysis. It computes for a given CPA D starting from (an) initial state(s) e_0 the set of reached states. Therefore it works on two sets abstract states, a list of states `reached` that have already been found to be reachable and a list of states `waitlist` that are not yet processed. The `waitlist` is iteratively processed by taking one state e in each iteration from the list and computing all of its successor states e' via the transfer relation. For each e' it is checked by the merge operator whether it is to be merged with an existing state e'' in the list `reached`. Then it is checked by the stop operator if the abstract successor state e' is already covered by a state in `reached`. If this is not the case, e' is added to both lists, `reached` and `waitlist`. There also exists a for dynamic precision adjustment augmented interface, the

Algorithm 1 CPA execution algorithm, taken form Ref. [23]

Input: a configurable program analysis $D = (D, \rightsquigarrow, \text{merge}, \text{stop})$, an initial abstract state $e_0 \in E$, let E denote the set of elements of the semi-lattice of D
Output: a set of reachable abstract states
Variables: a set reached of elements of E, a set waitlist of elements of E

```
1: reached := $\{e_0\}$
2: waitlist := $\{e_0\}$
3: while waitlist $\neq \emptyset$ do
4:     pop $e$ from waitlist
5:     for all $e'$ with $e \rightsquigarrow e'$ do
6:         for all $e'' \in$ reached do
7:             //Combine with existing abstract state.
8:             $e_{new}$ := merge($e'$,$e''$)
9:             if $e_{new} \neq e''$ then
10:                waitlist := (waitlist $\cup$ $\{e_{new}\}$)$\setminus\{e''\}$
11:                reached := (reached $\cup$ $\{e_{new}\}$)$\setminus\{e''\}$
12:            end if
13:        end for
14:        if $\neg$stop($e'$,reached) then
15:            waitlist := waitlist $\cup$ $\{e'\}$
16:            reached := reached $\cup$ $\{e'\}$
17:        end if
18:    end for
19: end while
20: return reached
```

extension CPA$^+$ $\mathbb{D}^+ = (D, \Pi, \rightsquigarrow, \text{merge}, \text{stop}, \text{prec})$ [28]. The precision Π defines the level of abstraction of the analysis, i.e., the set of variables that are tracked in the analysis. That means Π determines how coarse the analysis is.

2.1.2 Location CPA

The reached program locations can be tracked in the Location CPA $\mathbb{L} = (D_{\mathbb{L}}, \rightsquigarrow_{\mathbb{L}}, \text{merge}_{\mathbb{L}}, \text{stop}_{\mathbb{L}})$ [18], [23]. The lattice $\mathcal{E}_{\mathbb{L}}$, on which the abstract domain $D_{\mathbb{L}} = (C, \mathcal{E}_{\mathbb{L}}, [[\cdot]])$ is based on, consists of the set L of program locations where each $l \in L$ maps to a location in the CFA. The transfer relation $\rightsquigarrow_{\mathbb{L}}$ gives for each program location $l \in L$ the successor location $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} l'$ if there exists an edge g in the CFA directed from l tp l'. The merge operator $\text{merge}_{\mathbb{L}}(l, l') = l'$ does not merge the abstract states if the control flow meets and the operator $\text{stop}(l, R)$ indicates termination if l is already contained in R.

2.1.3 BDD-Based Program Analysis

A binary decision diagram (BDD) is a data structure for the representation of a boolean function based on the Shannon expansion [29], [30]. It is a rooted, directed, acyclic graph, where each non-terminal vertex is associated to a boolean variable v_n and has two outgoing edges. The lower one refers to the situation where v_n has the value 0 and the upper one to $v_n = 1$. Correspondingly, there are two types of leafs, the '0-terminal' and the '1-terminal'. A BDD is ordered if the variables occur always in the same order going from the root to the leafs and the BDD can be reduced by merging any isomorphic sub-graphs and eliminating any node whose two children are isomorphic. In the following all BDDs that are considered are reduced ordered BDDs.

By using BDDs as main data structure for representation of state sets in program analysis, not only boolean variables but also integer variables have to be considered [30]. The representation of data states of integers is achieved by encoding the integer assignments as bit vectors and integer variables as vectors of boolean variables.

A BDD-based program analysis is implemented in CPAchecker [17] as a configurable program analysis $\mathbb{BDD} = (D_{\mathbb{BDD}}, \rightsquigarrow_{\mathbb{BDD}}, \text{merg}_{\mathbb{BDD}}, \text{stop}_{\mathbb{BDD}})$, with the abstract domain $D_{\mathbb{BDD}} = (C, \mathcal{E}, [[\cdot]])$. Each abstract state $e \in E_{\mathbb{BDD}}$ contained in the lattice $\mathcal{E}_{\mathbb{BDD}} = (E_{\mathbb{BDD}}, \sqsubseteq)$ consists of a BDD to which the concretization function $[[\cdot]]$ assigns the set C of all concrete states that are represented by the BDD. The transfer relation $\rightsquigarrow_{\mathbb{BDD}}$ computes all abstract successor states e' for a given abstract state e traversing along all outgoing CFA edges g containing the program operation op: $e \overset{g}{\rightsquigarrow}_{\mathbb{BDD}} e'$. The resulting successor state e' has the form:

$$e' = \begin{cases} e \wedge e_{[p]} & \text{if } op = \texttt{assume(}p\texttt{)} \\ (\exists w : e) \wedge e_{[w=expr]} & \text{if } op = \texttt{(w := expr)}, \end{cases}$$

where e_φ is a BDD that is constructed from the formula φ. This means in the case of an assumption the BDD of the successor state is formed by conjugating the current BDD with a BDD that encodes the assumption. In case of an assignment, the current BDD is at first processed by existential quantification of the variable that gets a new value assigned and is then conjugated with a BDD representing the new value of the variable resulting in the BDD of the successor state. The merge operator is defined by $\text{merge}_{\mathbb{BDD}}(e, e') = e \vee e'$ and the termination check by $\text{stop}_{\mathbb{BDD}}(e, R) = \exists e' \in R : e \sqsubseteq e'$.

The purely BDD-based program analysis has turned out to be efficient for a special sub-class of event-condition-action programs which consist of a single loop in which many conditional branches occur because at the one hand all required operations appearing in these programs are efficiently supported by BDDs and at the other hand, these programs are challenging for most traditional analysis techniques as they have a complex control and data flow [30].

2.1.4 Predicate CPA

In predicate based program analysis the abstract states are represented using predicates over program variables. A predicate-based program analysis is implemented in CPAchecker [17] as extended configurable program analysis $\mathbb{P} = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \text{merge}_{\mathbb{P}}, \text{stop}_{\mathbb{P}}, \text{prec}_{\mathbb{P}})$ which implements the extended interface CPA$^+$ described in Section 2.1.1. The abstract domain $D_{\mathbb{P}} = (C, \mathcal{E}_{\mathbb{P}}, [[\cdot]])$ contained in $\mathbb{P}$ consists of the set C of concrete states, semi-lattice $\mathcal{E}_{\mathbb{P}}$ over abstract states $e \in E_{\mathbb{P}}$, and the concretization function $[[\cdot]]$ under the precisions $\Pi_{\mathbb{P}}$. The transfer relation $\rightsquigarrow_{\mathbb{P}}$ computes the abstract successor states e' for abstract state e under a precision π for an outgoing CFA edge g containing the program operation *op*: $e \overset{g}{\rightsquigarrow}_{\mathbb{P}} (e', \pi)$, with $e' = e \wedge e_{op}$, where e_{op} is a formula describing *op*. The merge operator $\text{merge}_{\mathbb{P}}$ combines states if their abstraction formula and location is the same and the termination check $\text{stop}_{\mathbb{P}}$ only checks coverage for abstraction states (states where the complete path formula is evaluated when SBE is not used). The precision-adjustment operator $\text{prec}_{\mathbb{P}}$ returns for an abstract state the state and precision or turns a non-abstraction state into an abstraction state. The precision $\pi \in \Pi_{\mathbb{P}}$ maps program locations to sets of predicates over program variables enabling the usage of different abstraction levels at different locations in the program. Typically, the initial precision is $\pi(l) = \emptyset$ for all locations $l \in L$. Dynamic precision adjustment is not used during the CPA algorithm in standard predicate CPA but enables the usage of $\mathbb{P}$ in the counterexample-guided abstraction refinement (CEGAR) [6], [7], [25]. CEGAR is a iterative algorithm for finding a suitable precision for the verification of a program that is fine enough to be able to report if the error location l_{ERR} can be reached but coarse enough to efficiently run the program analysis. The algorithm (Algorithm 2) starts with a coarse initial precision π_0 that only tracks few relations. With π_0 the underlying program analysis CPA^+ (extended form of Algorithm 1) is executed. If the CPA finds l_{ERR} under the given precision not reachable, the algorithm stops and reports that the program is safe. Otherwise, if the CPA^+ has found an execution path of the program where l_{ERR} is reached there are two possibilities: Either the found error path is an actual feasible path which means it is a concrete program execution or the error path is a infeasible path that only had been found because of the imprecise abstraction. In the first case, the algorithm stops and reports the found error path as counterexample. In the second case, the precision is refined and the algorithm restarted. Such a refinement procedure extracts information from infeasible error path and returns a precision π that would indicate the infeasibility of this error path.

2.2 Backwards Model Checking

The model checking approach described in the previous sections analyses programs in a forward direction. In principle, model checking can also be done in a backwards manner. Backwards model checking has already been applied in hardware model checking [8] as well as in symbolic software model checking based on BDDs [9]. Also combinations of forward and backward reachability analysis have been applied in hardware model checking [10], software verification with BDDs [11]–[13], abstract interpretation [14] or using Craig interpolants [15], [16]. A reachability analysis as described in section 2.1 can also be done backwards by constructing an ARG. In this case, the construction starts at the investigated program location l_{ERR}. The backwards ARG (bwARG) is build up by successively traversing the edges of the CFA in reverse direction of arrow. A successor state here is obtained by the weakest precondition of the current state and the operation of the traversed edge. The iterative buildup stops if all leafs contain a path formula of '*false*' or there exists an ARG node which contains the entry of the program l_{ENTRY} as location and a non-contradicting path formula. The first case proves as in the forward ARG that there is no program execution path in which l_{ERR} can be reached. The latter case reports an execution path in which l_{ERR} is reached going from the leaf containing l_{ENTRY} to the root of the bwARG. Considering again the example program of section 2.1 (Example 1) with CFA (Figure 1) the construction of a bwARG (Figure 3) starts at the error label location (line 7) and the successor node is

Algorithm 2 CEGAR($\mathbb{D}^+, e_0, \pi_0$), taken form Ref. [7]

Input: CPA with dynamic precision adjustment $\mathbb{D} = (D, \Pi, \mathrm{merge}, \mathrm{stop}, \mathrm{prec})$, initial abstract state $e_0 \in E$ with precision $\pi \in \Pi$, where E denotes the set of elements of the semi-lattice of D

Output: verification result *safe* or *unsafe*

Variables: set `reached` $\subseteq E \times \Pi$, set `waitlist` $\subset E \times \Pi$, error path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$

```
1: reached := $\{(e_0, \pi_0)\}$;
2: waitlist := $\{(e_0, \pi_0)\}$;
3: $\pi := \pi_0$;
4: while true do
5:     (reached,waitlist) := CPA$^+$($\mathbb{D}$,reached,waitlist);
6:     if waitlist = $\emptyset$ then
7:         return safe
8:     else
9:         $\sigma$ := extractErrorPath(reached);
10:        if isFeasible($\sigma$) then //error path is feasible: report bug
11:            return unsafe
12:        else//error path is not feasible: refine and restart
13:            $\pi := \pi \cup$ Refine($\sigma$);
14:            reached := $(e_0, \pi)$;
15:            waitlist := $(e_0, \pi)$;
16:        end if
17:    end if
18: end while
```

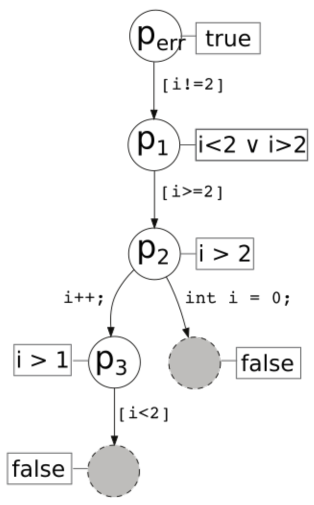


Figure 3: Backwards ARG for the example program Example 1.The path formula of the respecting abstract state (node) is attached in a box. The edges contain as the CFA (Figure 1) the program statements.

computed by interpretation of the incoming CFA edge '`[i != 2]`'. Here again, the complete

ARG has only leafs containing '*false*' revealing that the error label in line 7 cannot be reached. On average, the forward and backward analysis should be equally powerful but comparing both ARGs, the one of the forwards analysis (Figure 2) and the backwards analysis (Figure 3), reveals that there are cases in which the backwards analysis needs less computations (less states). It remains to be investigated whether there is a class of programs on which the backwards analysis performs better than the forward analysis.

Looking again at the example bwARG (Figure 3) makes clear that a backwards program analysis only is reasonable for analyses that memorize the change history of variables rather than storing only exact values like in explicit-value analyses [31]. In a explicit-value analysis `i` would stay undefined until the analysis reaches line 2 of Example 1 and thus would not be able to give a verification result. Such a situation would arise for many programs since the assignment of a value to a variable is naturally at the beginning of the program. For this reason the symbolic program analyses in the BDD and predicate domain are investigated for the use in backwards direction.

2.2.1 Backwards CPA

For a given CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$ the corresponding backwards CPA $\mathbb{D}_{bw} = (D, \rightsquigarrow_{bw}, \text{merge}, \text{stop})$ has the same abstract domain D and same operators merge and stop. The only difference in $\mathbb{D}$ and $\mathbb{D}_{bw}$ is the calculation of the successor states meaning $\mathbb{D}$ and $\mathbb{D}_{bw}$ only differ in the transfer relation. Backwards CPAs are executed by the same algorithm than forwards CPAs (see Algorithm 1). The same holds for the extension CPA$^+$.

2.2.2 Backwards Location CPA

As described in Section 2.1.2, the transfer relation $\rightsquigarrow_{\mathbb{L}}$ of the Location CPA $\mathbb{L}$ in forwards analysis direction gives at the program location l for the outgoing CFA edge g with $g = (l, op, l')$ the successor state l': $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} l'$. The transfer relation $\rightsquigarrow_{\mathbb{L}_{bw}}$ contained in the backwards Location CPA $\mathbb{L}_{bw} = (D_{\mathbb{L}}, \rightsquigarrow_{\mathbb{L}_{bw}}, \text{merge}_{\mathbb{L}}, \text{stop}_{\mathbb{L}})$ is defined as the inverse function of $\rightsquigarrow_{\mathbb{L}}$. This means $\rightsquigarrow_{\mathbb{L}_{bw}}$ gives at location l' for the incoming edge $g = (l, op, l')$ the successor location l: $l' \overset{g}{\rightsquigarrow}_{\mathbb{L}_{bw}} l$.

2.2.3 Backwards BDD CPA

The backwards BDD CPA $\mathbb{BDD}_{bw} = (D_{\mathbb{BDD}}, \rightsquigarrow_{\mathbb{BDD}_{bw}}, \text{merg}_{\mathbb{BDD}}, \text{stop}_{\mathbb{BDD}})$ differs from the corresponding BDD CPA $\mathbb{BDD}$ in forwards analysis direction only in the transfer relation. The transfer relation $\rightsquigarrow_{\mathbb{BDD}_{bw}}$ of $\mathbb{BDD}_{bw}$ calculates the successor abstract state e' for abstract state e by interpretation of the program operation op contained in the incoming CFA edge g: $e_{bw} \overset{g}{\rightsquigarrow}_{\mathbb{BDD}_{bw}} e'_{bw}$, with

$$e'_{bw} = \begin{cases} e_{bw} \wedge e_{[p]} & \text{if } op = \texttt{assume(}p\texttt{)} \\ \exists w : (e_{bw} \wedge e_{[w=expr]}) & \text{if } op = \texttt{(w := expr)}. \end{cases}$$

In the case of $op = \texttt{assume(}p\texttt{)}$ the interpretation is the same as for the forwards transfer relation. Thus it is decoded in exactly the same BDD $e_{[p]}$ in both analysis directions. The interpretation of $op =$ `w := expr` in the backwards analysis slightly differs from the interpretation in the forwards BDD analysis. In the backward analysis the order of the sub-steps of the calculation of the successor BDD is reversed. Here, at first the BDD e_{bw}

of the current state is combined by conjunction with a BDD representing the new variable assignment $e_{[w:=expr]}$. Substantially, w is existentially quantified, as the value of a variable is not yet determined in the locations preceding its assignment. In the special case, where the assigned variable w also occurs right-hand side, the assignment statement is split into two statements by first assigning the old value of w to a temporary variable. This can be best illustrated using a example program cutout (see Example 2) and the corresponding CFA (Figure 4).

Example 2 Code snippet for illustration of the interpretation of assignments in backwards analysis direction.

```
1: a = 3;
2: a = a + 1;
3: if a == 2 then
4:     ERROR: return 1;
5: end if
```

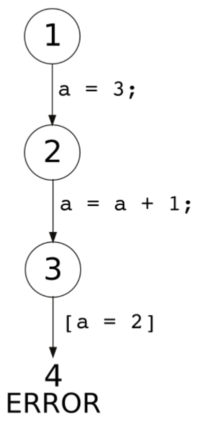


Figure 4: Control flow automaton of the example program cutout (Example 2). The nodes represent program locations with the respecting line number as label. The edges represent program statements. Assumptions like `assume(p)` are denoted as `[p]`.

The backwards analysis starts at the error label in line 4 of the example program with a BDD $e[1] = e_{true}$ representing '*true*'. The first statement that is evaluated is the assumption `a == 2` in line 3, so the state is represented by the BDD $e[2] = e[1] \wedge e_{[a=2]} = e_{[a=2]}$. Next, the analysis comes to the assignment `a = a + 1`. Here the variable occurs left-hand side as well as right-hand side. So the assignment is split into:

$$\texttt{a = a + 1} \begin{cases} \texttt{tmp = a} \\ \texttt{a = tmp + 1} \end{cases}$$

for the interpretation. As the analysis direction is backwards, `a = tmp + 1` is interpret fist according to the interpretation rules for an assignment. This gives the temporary BDD

$$e_{\text{TMP}} = \exists a : (e[2] \wedge e_{[a=tmp+1]}) = \exists a : e_{[a=2 \wedge a=tmp+1]} = e_{[tmp=1]}.$$

Subsequently, the temporary variable disappears again in the interpretation by evaluating `tmp = a`:

$$e[3] = \exists tmp : (e_{\text{TMP}} \wedge e_{[tmp=a]}) = \exists tmp : e_{[tmp=1 \wedge tmp=a]} = e_{[a=1]}.$$

Last, the analysis approaches `a = 3`, which results in a BDD containing a contradiction:

$$e[4] = \exists a : (e[3] \wedge e_{[a=3]}) = \exists a : e_{[a=1 \wedge a=3]} = \exists a : e_{false} = e_{false}.$$

2.2.4 Backwards Predicate CPA

The corresponding backwards CPA to $\mathbb{P}$ in the predicate domain is defined by $\mathbb{P}_{bw} = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}_{bw}}, \text{merge}_{\mathbb{P}}, \text{stop}_{\mathbb{P}}, \text{prec}_{\mathbb{P}})$. $\mathbb{P}_{bw}$ is distinguished from $\mathbb{P}$ only by the transfer relation $\rightsquigarrow_{\mathbb{P}_{bw}}$. De facto, the successor state e'_{bw} given by $e_{bw} \overset{g}{\rightsquigarrow}_{\mathbb{P}_{bw}} e'_{bw}$ has a similar form as the successor state obtained by $\rightsquigarrow_{\mathbb{P}}$ (see Section 2.1.4): $e'_{bw} = e_{bw} \wedge e_{op}$, where the interpretation e_{op} of op is for both analysis directions similar (for assume and assignment statements).

Assumption $[p]$ is interpreted in both cases by augmenting the path formula by the assumption: $e' = e \wedge [p]$ and $e'_{bw} = e_{bw} \wedge [p]$, at which in both analysis directions the current static single assignment (SSA) indices of the occurring program variables are used. Also for an assignments $w := expr$, in both analysis directions this information is appended to the path formula. The interpretation of assignments only differs in the used SSA indices for the assigned variables. In the forward analysis, the value $expr$ is assigned in the assignment $w := expr$ to variable w with a fresh index: $w[n+1] = expr$. If in $expr$ also variable w is contained, the old index n is used for this occurrence of w. In the backwards analysis in contrast, the value $expr$ is assigned to the variable w with the current index $w[n] = expr$ while a fresh index $n+1$ is used for w if it occurs in $expr$. So the new path formulas would be: $e' = e \wedge (w[n+1] = expr)$ and $e'_{bw} = e_{bw} \wedge (w[n] = expr)$. This situation is visualized considering Example 2 again.

The forward analysis starts at the beginning of the program, so the first statement that is interpreted is `a = 3` (line 1) as path formula $e[1] = (a[1] = 3)$. The next statement `a = a + 1` contains variable `a` on the left-hand side as well as the right-hand side of the assignment. Here, the augmented 'old' value of `a` is reassigned resulting in a new value for `a`. The new path formula hence is $e[2] = (a[1] = 3) \wedge (a[2] = a[1] + 1)$. The last statement is an assume statement `[a =  2]`. In its interpretation no new index for `a` is created as the value of `a` remains the same in the program execution. So the resulting path formula of the forward analysis for the whole program fragment is:

$$e[3] = (a[1] = 3) \wedge (a[2] = a[1] + 1) \wedge (a[2] = 2),$$

which contains a contradiction and would result in '*false*'. Performing a backwards analysis on the same program code, the analysis starts at the error label in line 4. So the assume statement `a = 2` is the first to be interpreted giving a path formula of $e_{bw}[1] = (a[1] = 2)$. Next, the analysis comes to the statement `a = a + 1`, where variable `a` appears at both sides of the assignment. As the left-hand side `a`, to which a new value is assigned, is 'the same' as the `a` in the previous assumption, the current index must be kept for the left-hand side `a` while the `a` appearing on the right-hand side in the assigned value is a 'different' `a`, so a fresh index has to be created for it. The resulting path formula is $e_{bw}[2] = (a[1] = 2) \wedge (a[1] = a[2] + 1)$. The last statement interpreted in the backwards analysis is `a = 3`, resulting in a path formula for the whole code fragment of

$$e_{bw}[3] = (a[1] = 2) \wedge (a[1] = a[2] + 1) \wedge (a[2] = 3),$$

which also gives '*false*'.

3 Implementation of Backwards Analyses in CPAchecker

3.1 Backwards CPA

Program analyses are implemented in CPAchecker [17] as CPAs (see Section 2.1.1) by the use of the composite Pattern (see Figure 5). The CPA execution algorithm operates on an object

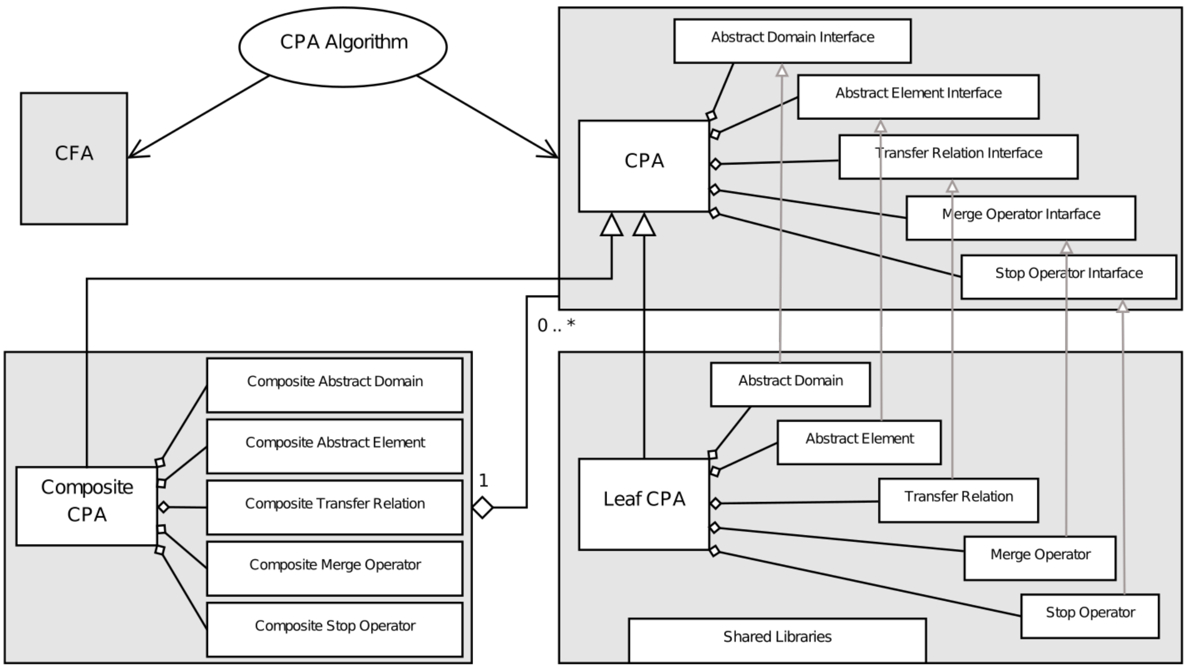


Figure 5: Core architecture of CPAchecker. Figure taken from Ref. [17].

of the CPA interface leaving the concrete specification of the CPA variable. The concrete CPA can be a Composite CPA which combines different CPA into one CPA. The CPAs contained in the Composite CPA can again be a Composite CPA or a Leaf CPA. A special form of the Leaf CPA is the Wrapper CPA which wraps around another CPA. This construction is used for realization of backwards analyses in CPAchecker. A Wrapper CPA *BackwardCPA* is implemented. It swaps the methods *getAbstractSuccessors*/*getAbstractSuccessorsForEdge* and *getAbstractPredecessor*/*getAbstractPredecessorsForEdge* of the *Transfer Relation Interface*. As the only difference between a CPA and its corresponding backwards CPA is the interpretation direction of the CFA edges in the transfer relation (see Section 2.2.1) this wrapping is sufficient for changing the analysis direction of any CPA that implements all of the methods specified above.

This framework for backwards program analysis was established for backwards BDD analysis by Friedberger *et al.* (not published). For unification of backwards analyses in CPAchecker, the already existing backwards predicate CPA, which didn't make use of the *BackwardCPA*, was refactored in this work so that the analysis direction of the predicate CPA can now be reversed using this Wrapper CPA.

3.2 Presentation of the Results

The results of a backwards analysis in CPAchecker are presented in a HTML interface like for the corresponding forwards analysis. In the HTML document, the CFA as well as the ARG are shown graphically. Additionally, it displays the source code of the verified program and the log messages of the analysis.

The visualization of the ARG and CFA is given in graphviz format (.dot) additionally to the graphical representation in the HTML document and in the file 'reached.dot', the CFA with the abstract states on top is given.

The counterexample export is also available for the backwards analysis. Here, the information about a found error-path is given in different formats including simple txt-format, C code, or graphically in dot-format.

4 Evaluation

For investigation whether there exists a class of programs on which the backward analysis performs more efficiently than the program analysis in forward direction the analysis in both directions was compared on several benchmark tasks.

4.1 Compared Verification Approaches

The impact of the analysis direction was investigated for the program analyses using BDD or predicates for the abstract domain. In the following, the forward analysis in the BDD domain is referred to as *BDD* while the corresponding backwards analysis is referred to as *bwBDD*. In the same way, the predicate analysis is named as *Pa* and *bwPa*. Additionally, the traversal order in the analysis is compared for both directions. If depth-first search is used insted of the default breadth-first search, the suffix *-DFS* is appended to the respecting name of the analysis approach.

4.2 Verification Tasks

For comparison of the forward and backward reachability analysis, verification tasks were taken from the verification task repository of SV-COMP 19 [21]. The sub-categories *BitVectorsReach*, *ControlFlow*, *ECA*, *HeapReach*, *Loops*, and *ProductLines* from *ReachSafety* were verified using the 'unreach-call' specification that checks for the not-reachability of a certain function call. These categories were chosen as they mainly contain simple loop structures and conditional branches. Sub-categories *ArraysReach*, *Recursive*, *Sentimentalized*, and the categories *Concurrency* and *Systems_DeviceDriversLinux64_ReachSafety* were excluded as they require pointer aliasing and recursive function call handling which is not yet supported by the backwards analyses.

4.3 Experimental Setup

The verification tasks were run using revision 31643 of the *backwardsTransfer* branch of CPAchecker [17] with the options:
```
- benchmark
- heap 5000M
- timelimit 90s
- stats.
```
The configuration of the *BDD* and the *bwBDD* analyses can be found in the configuration files `bddAnalysis.properties` and `bddAnalysis-backwards.properties`, respectively. For *Pa* an additional option of `cpa.predicate.handlePointerAliasing = false` was used extending the configuration of `predicateAnalysis.properties`. This option was applied to disable the pointer aliasing handling for a better comparability to *bwPa* configured in `predicateAnalysisBackward.properties` (renamed to `predicateAnalysis-backwards.properties` in the current revision) as here this option is not yet supported.

Here it is worth mentioning that for the forwards analyses, the initial state is the entrance of the main function, while it is the defined error location l_{ERR} for the backwards analyses. This is specified in the configuration files by the option `analysis.initialStatesFor`, which is set to `ENTRY` or `TARGET`, respectively. The reachability of the defined error location is monitored with a different automaton in both analysis directions. In the forwards analyses the automaton, which is defined in `config/specification/sv-comp-reachability.spc`, is used by the option `specification`. This automaton indicates the reaching of l_{ERR} by switching from the state 'Initial' to the error state if the analysis comes over the function call `__VERIFIER_error()`.

In the backwards analysis the automaton `config/specification/MainEntry.spc`, specified in the option `backwardSpecification`, is used for checking the reachability of l_{ERR}. Since in backwards analysis direction a location l_{ERR} is reachable if the analysis reaches the beginning of the program starting from l_{ERR}, this automaton switches from the initial state 'Body' to state 'MainEntry' if the evaluated line matches the regular expression of the head of a main function and further switching to an error state indicating that the entry point of the main function is reached which means that l_{ERR} can be reached in an execution of the program (see Figure 6).

Figure 6: Automaton specified in `MainEntry.spc`. This automaton detects the entry point of the main function.

```
OBSERVER AUTOMATON MainEntryAutomaton

INITIAL STATE Body;

STATE USEFIRST Body :
   MATCH [.*\\s+main\\s*\\(.*\\)\\s*;?.*] -> GOTO MainEntry;

STATE USEFIRST MainEntry :
   MATCH ENTRY -> ERROR("main entry reached");

END AUTOMATON
```

The analyses with depth-first traversal were performed using the option `analysis.traversal.order = DFS`.

All tasks were run on machines with a Intel Core i5-4590 (3.30 GHz) that have 33 GB of RAM and a memory limitation of 7 GB.

4.4 Results

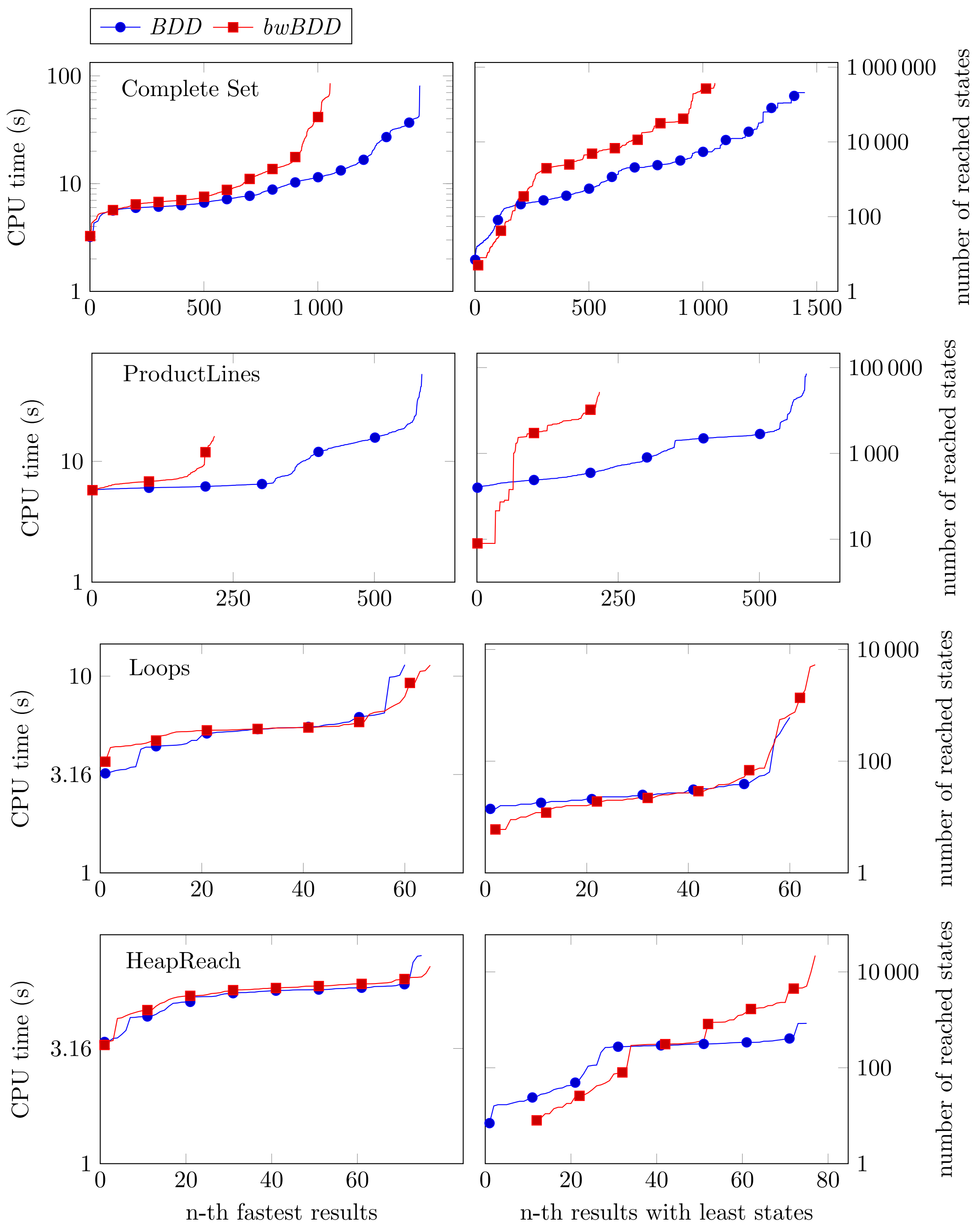
4.4.1 *BDD* vs. *bwBDD*

Comparing the verification approaches *BDD* and *bwBDD* on the benchmark set defined in 4.2 reveals that both give similar results whereat the forwards analysis solves slightly more tasks successfully (see Table 1). The main difference is the number of correct results of '*false*'. This difference primarily arises from the subset *ProductLines*. Here, *BDD* gives for 262 a correct result of '*false*' whereas *bwBDD* has only found for 6 tasks a correct '*false*' as it runs out of memory or results in a timeout in most cases. Both approaches give for about the same number of verification tasks incorrect results. Also both analysis approaches do not exhibit big differences in performance. The required CPU time for the correctly solved tasks (Figure 7) is approximately the same for both approaches. The *bwBDD* analyses

Table 1: Detailed results of the *BDD* and *bwBDD* analysis.

		correct results		incorrect results	
Benchmark subset		*BDD*	*bwBDD*	*BDD*	*bwBDD*
Complete set (2449 tasks)	total	1448	1054	558	530
	true	692	600	2	2
	false	756	454	556	528
ProductLines (597 tasks)	total	584	217	10	0
	true	322	211	0	0
	false	262	6	10	0
Loops (208 tasks)	total	60	65	46	44
	true	23	35	0	0
	false	37	30	46	44
HeapReach (241 tasks)	total	75	77	123	111
	true	7	17	0	0
	false	68	60	123	111
ECA (1256 tasks)	total	641	616	325	325
	true	306	306	0	0
	false	335	310	325	325
ControlFlow (95 tasks)	total	71	62	21	17
	true	30	27	0	0
	false	41	35	21	17
BitVectors Reach (52 tasks)	total	17	17	33	33
	true	4	4	2	2
	false	13	13	31	31

however, has for the majority of the tasks more reached states than *BDD*. This again issues from the *ProductLines* subset but also to some extent from subset *ECA*. In addition it has to be noted that for the *HeapReach* subset, the backwards analysis reaches for some tasks zero states. The reason for this could be that the initial location l_{ERR} is not found due to differences in the notation of the error label or the absence of an error label.


BDD
bwBDD
Complete Set
ProductLines
Loops
HeapReach
CPU time (s)
number of reached states
n-th fastest results
n-th results with least states

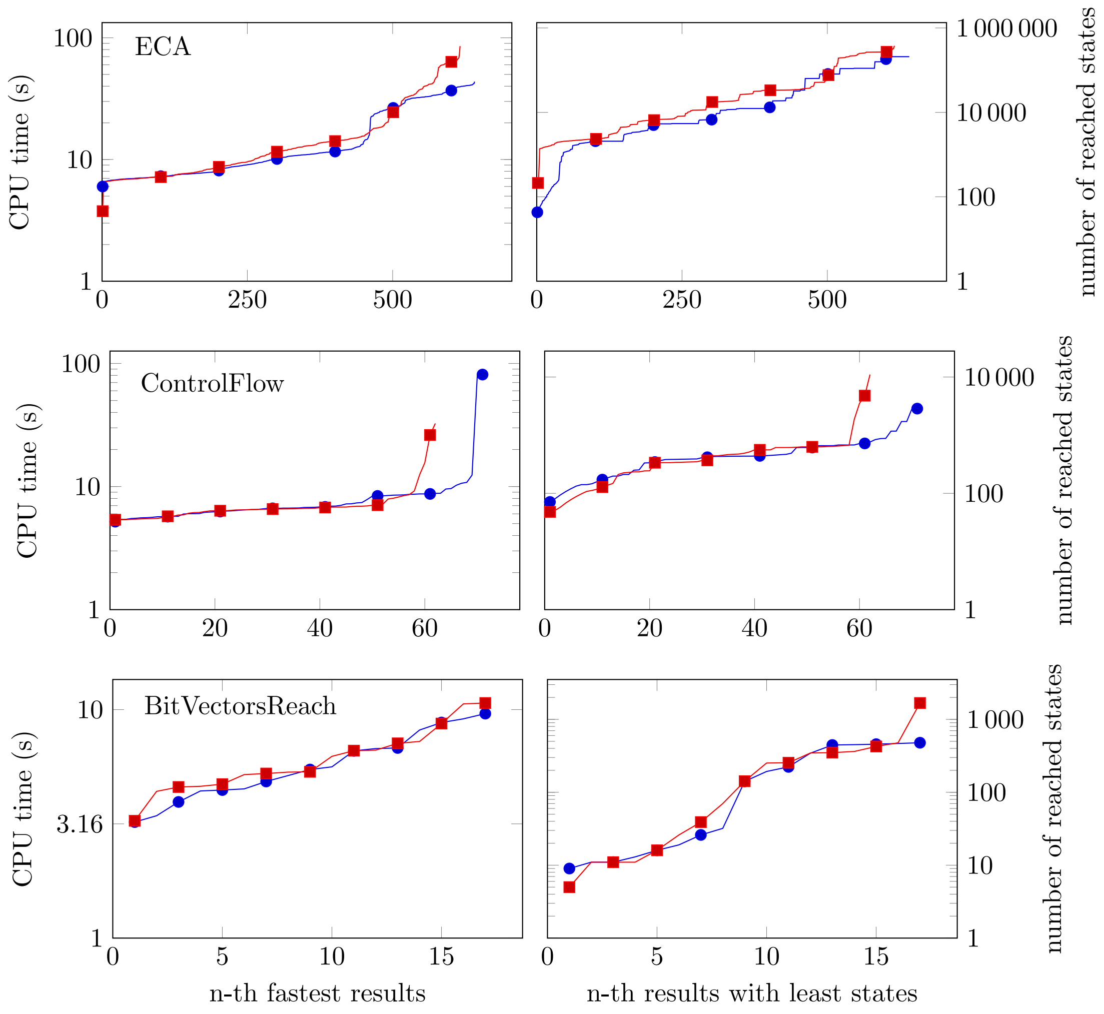


Figure 7: Quantile functions CPU time (left) and number of reached states (right) needed for the verification tasks of the complete benchmark set described in Section 4.2 and the subsets of it for the *BDD* (blue) and *bwBDD* (red) analysis. Only tasks are plotted for which the correct verification result was found.

4.4.2 *BDD-DFS* vs. *bwBDD-DFS*

Table 2: Detailed results of the *BDD-DFS* and *bwBDD-DFS* analysis.

		correct results		incorrect results	
Benchmark subset		*BDD-DFS*	*bwBDD-DFS*	*BDD-DFS*	*bwBDD-DFS*
Complete set (2449 tasks)	total	1447	1054	558	530
	true	692	600	2	2
	false	755	454	556	528
ProductLines (597 tasks)	total	584	217	10	0
	true	322	211	0	0
	false	262	6	10	0
Loops (208 tasks)	total	60	65	46	44
	true	23	35	0	0
	false	37	30	46	44
HeapReach (241 tasks)	total	75	77	123	111
	true	7	17	0	0
	false	68	60	123	111
ECA (1256 tasks)	total	641	616	325	325
	true	306	306	0	0
	false	335	310	325	325
ControlFlow (95 tasks)	total	70	62	21	17
	true	30	27	0	0
	false	40	35	21	17
BitVectors Reach (52 tasks)	total	17	17	33	33
	true	4	4	2	2
	false	13	13	31	31

The usage of depth-first search (DFS) instead of bread-first search (BFS) in the BDD analysis does not have an impact on either the accuracy or on the performance of the analysis in both analysis directions. They both give the same verification results (compare Table 1 and 2) requiring the same amount of CPU time and number of reached states (data not shown).

4.4.3 *Pa* vs. *bwPa*

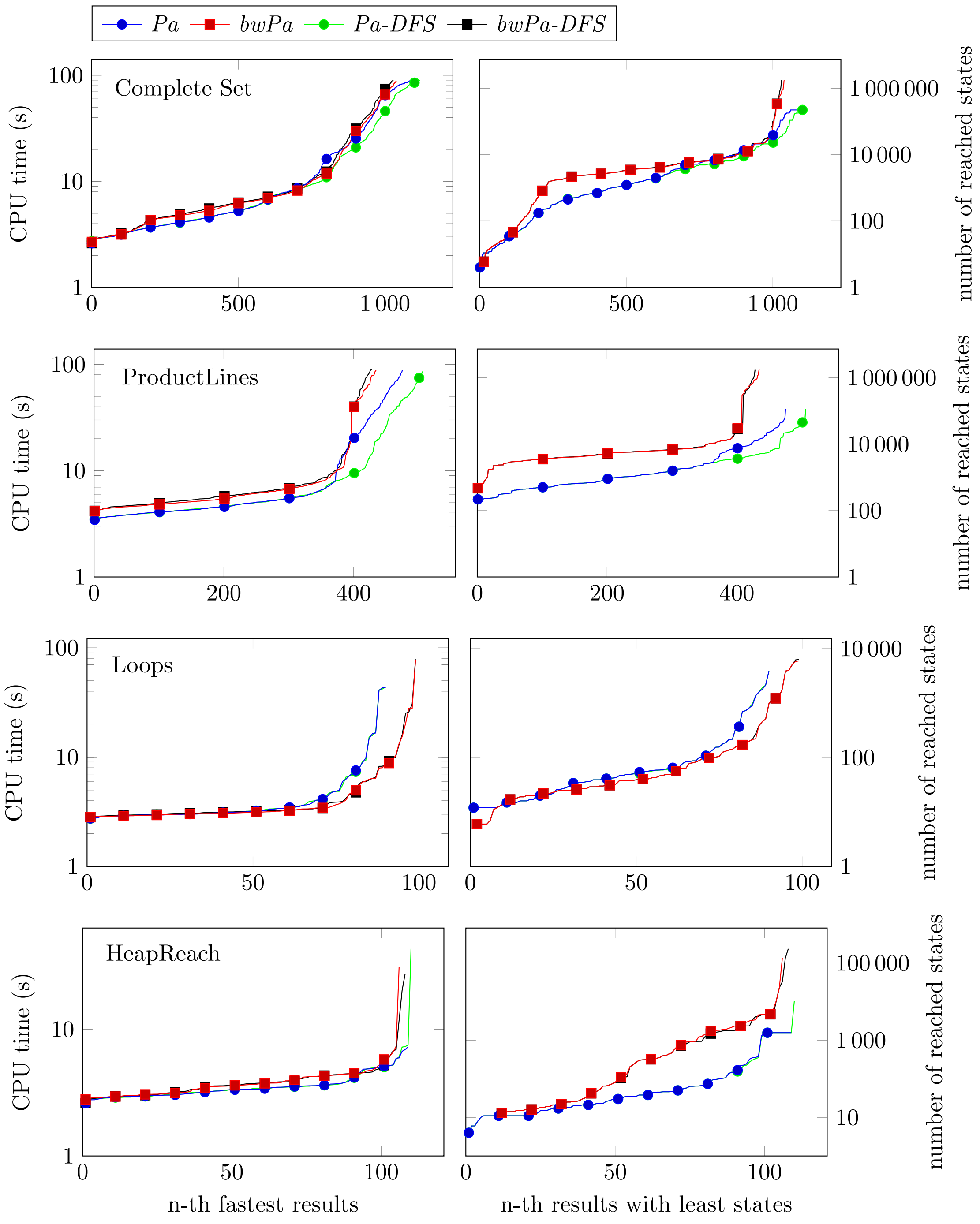
Table 3: Detailed results of the *Pa* and *bwPa* analysis.

		correct results		incorrect results	
Benchmark subset		*Pa*	*bwPa*	*Pa*	*bwPa*
Complete set (2449 tasks)	total	1090	1039	144	141
	true	692	718	16	16
	false	398	321	128	125
ProductLines (597 tasks)	total	475	435	0	0
	true	290	287	0	0
	false	185	148	0	0
Loops (208 tasks)	total	90	99	18	14
	true	56	65	4	4
	false	34	34	14	10
HeapReach (241 tasks)	total	109	106	125	126
	true	47	45	11	11
	false	62	61	114	115
ECA (1256 tasks)	total	296	297	0	0
	true	231	268	0	0
	false	65	29	0	0
ControlFlow (95 tasks)	total	86	72	0	0
	true	46	34	0	0
	false	40	38	0	0
BitVectors Reach (52 tasks)	total	34	30	1	1
	true	22	19	1	1
	false	12	11	0	0

Comparing the predicate analysis for forward and backward analysis direction reveals that both are almost identical regarding the verification result (Table 3). Even for the *ProductLines* subset, for which the backwards BDD analysis performed significantly inferior, *Pa* and *bwPa* verify nearly the same number of programs successfully. Both approaches give some incorrect results for the *Loops* and *HeapReach* subset which is in most cases due to the disabled pointer aliasing handling.

Here again, both approaches need the same CPU time for the correctly verified programs (see Figure 8, top left). In total, however, the performance of *bwPa* is to some extent inferior to *Pa* as it has on average more reached states (Figure 8, top right). This is clearly visible for the *ProductLines*, *HeapReach* subsets but also holds for *ControlFlow* and *BitVectorsReach*. Only *ECA* doesn't follow this trend. Here, the backwards analysis has less reached states and needs less CPU time for the correctly verified tasks.

As the *bwBDD* analysis, *bwPa* has for some tasks of the *HeapReach* subset zero reached states. This is not surprising as the same statement as initial location of the analysis is used for both backwards analyses (for further explanations see Section 4.4.1).



Pa
bwPa
Pa-DFS
bwPa-DFS
Complete Set
ProductLines
Loops
HeapReach
CPU time (s)
number of reached states
n-th fastest results
n-th results with least states

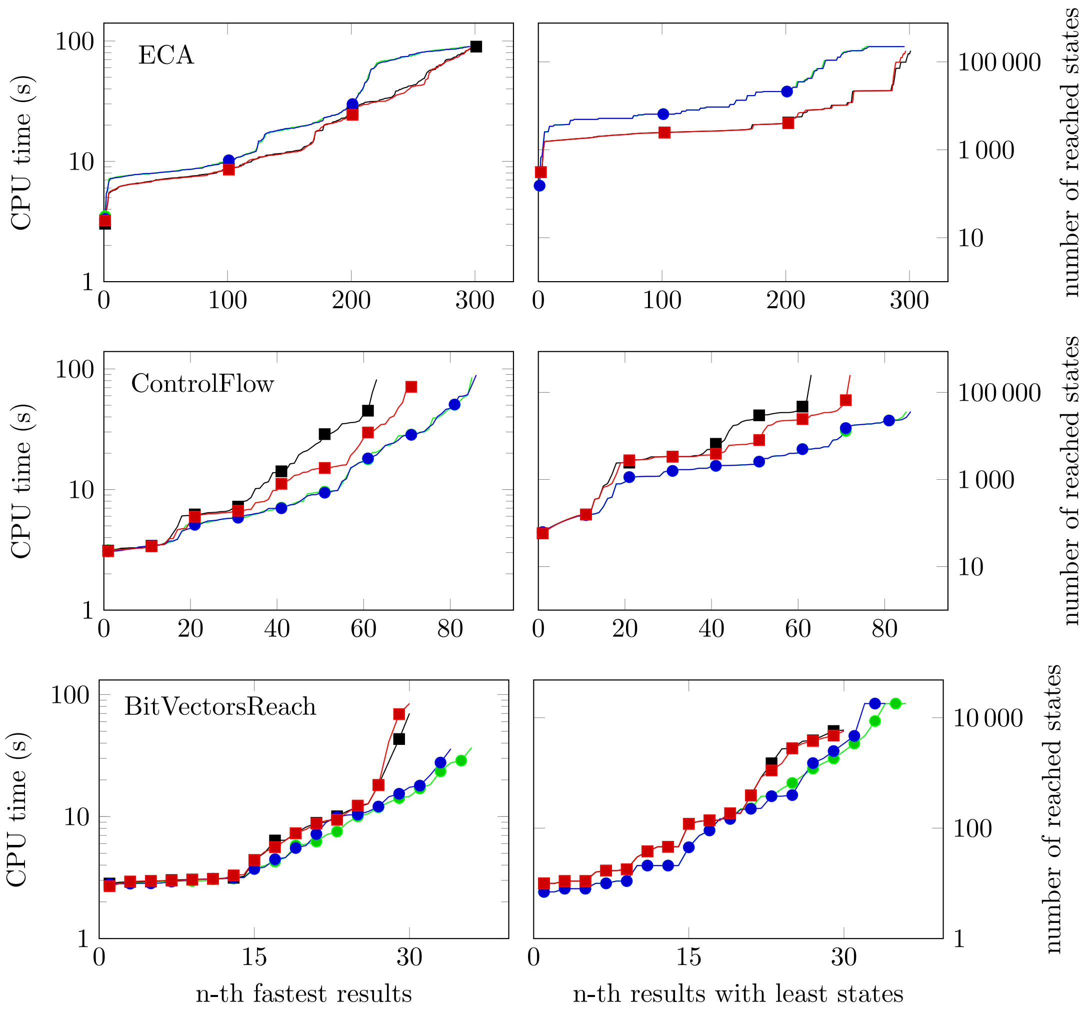


Figure 8: Quantile functions regarding the CPU time (left) and number of reached states (right) needed for the verification tasks of the complete benchmark set described in Section 4.2 and the subsets of it for the *Pa* (blue) and *bwPa* (red) analysis. The results of the approaches with DFS traversal order are plotted in green (*Pa-DFS*) and black (*bwPa-DFS*) but are covered by the respecting BFS analysis in most cases as they give similar results. Only tasks are plotted for which the correct verification result was found.

4.4.4 ***Pa-DFS*** vs. ***bwPa-DFS***

Table 4: Detailed results of the *Pa-DFS* and *bwPa-DFS* analysis.

		correct results		incorrect results	
Benchmark subset		*Pa-DFS*	*bwPa-DFS*	*Pa-DFS*	*bwPa-DFS*
Complete set (2449 tasks)	total	1120	1029	145	144
	true	696	722	16	16
	false	424	307	129	128
ProductLines (597 tasks)	total	506	428	0	0
	true	296	287	0	0
	false	210	141	0	0
Loops (208 tasks)	total	90	99	18	17
	true	56	65	4	4
	false	34	34	14	13
HeapReach (241 tasks)	total	110	108	126	126
	true	47	45	11	11
	false	63	63	115	115
ECA (1256 tasks)	total	293	301	0	0
	true	228	270	0	0
	false	65	31	0	0
ControlFlow (95 tasks)	total	85	63	0	0
	true	45	36	0	0
	false	40	27	0	0
BitVectors Reach (52 tasks)	total	36	30	1	1
	true	24	19	1	1
	false	12	11	0	0

In the predicate analysis, in contrast to the BDD analysis, the change of the traversal order from BFS to DFS does make a slight difference. While for *Pa-DFS* the number of correct results increases with the DFS option from 1090 to 1120 it decreases from 1039 to 1029 for *bwPa-DFS* compared to *bwPa* (compare Table 3 and 4). This difference arises mainly from the subset *ProductLines* and for the backwards analysis additionally from the *ControlFlow* subset. The number of incorrect results stays for both analysis direction about the same. The performance regarding CPU time and reached states, however, is about the same as for *Pa* and *bwPa* (see Figure 8).

4.5 Conclusion

For the benchmark set used in this work, the backwards analysis is in most cases outperformed by the respecting forwards analysis. In the *Complete Set*, the forwards analyses verify more tasks correctly than the respecting backwards analyses. An exception is the *Loops* subset where *bwBDD* as well as *bwPa* give slightly more correct results than *BDD* or *Pa*.

The numerous incorrect results in the *Pa* as well as the *bwPa*, especially in the *HeapReach* subset, are owed the disabled handling of pointer aliasing. This option was chosen for the comparability of both analysis directions as this option is not yet available for the backwards analysis.

The performance of forwards and respecting backwards analysis regarding CPU time is about the same for both, predicate analysis and BDD analysis.

The performance regarding the number of reached states, however, is in most cases inferior in the backwards analysis compared to the forwards analysis. This could be due to a unfavorable selection of test programs for backwards analysis or a not yet detected bug in the implementations of the backwards analysis.

A change of traversal order from BFS to DFS apparently does not have an impact on the BDD analysis in both analysis directions at all. For the predicate analysis the DFS traversal does make a little difference. Here, the number of correct verified results slightly increases for *Pa* in the *ProductLines* subset and slightly decreases for *bwPa* in the *ProductLines* and *ControlFlow* subsets while for both the performance does not change.

Summing up, the results indicate that the backwards analyses, *bwBDD* and *bwPa*, as they are currently implemented in CPAchecker, perform not as good as the respecting forwards analysis but there are some types of programs (*Loops* subset and *ECA* for *Pa*) where the backwards analysis is as good as or even better than the respecting forwards analysis.

5 Outlook

The backwards predicate analysis could be further improved by enabling the pointer aliasing handling also for the backwards analysis direction. As pointer aliasing is a common pattern in C programming language, this is an important feature for a competitive program analysis. Unfortunately, the pointer aliasing handling, as it is currently implemented for the forwards predicate analysis in CPAchecker [17], is not applicable for backwards analysis. Here, after the occurrence of a referencing `p = &a`, `a` and `*p` are treated as aliases. In the remaining analysis, if one of the aliases occurs in the left-hand side of an assignment, the other one is considered as well. In the backwards analysis this concept doesn't work. If here the analysis

Example 3 Code snippet for illustration of the pointer aliasing handling in the predicate analysis as it is currently implemented in CPAchecker [17].

```
1: a = 3;
2: p = &a;
3: a = 1;
4: *p = 5;
5: *p = 0;
6: if a == 0 then
7:     ERROR: return 1;
8: end if
```

comes over a referencing, the pointer aliasing hasn't to be handled any more since in the code lines preceding the referencing the variables aren't aliased. The pointer aliasing can also not be considered repressively either as the history in which the two variables have been assigned is not saved. This is best illustrated on an example (see Example 3). Analyzing the example program in a backwards analysis, first it is assumed $a[1] = 0$. Then lines 5 and 6 are interpreted as $*p[1] = 0$ and $*p[2] = 5$. Next, line 4 gives $a[1] = 1$. Then the analysis comes to the referencing `p = &a` in line 2. The information about this aliasing is no longer needed in the remaining analysis, as the variables aren't aliased in the lines above. The aliasing can also not be handled afterwards for the already interpreted lines as the history in which `a` and `*p` were accessed has not been memorized and thus it is not possible to decide which `a` resembles which of the `*p`.

The handling of pointer aliasing would have to be adapted for the backwards analysis in a way that it somehow considers all possible variables as aliased location as soon as the analysis comes over an assignment of a variable over a pointer such as `*p = 0` in line 5 of Example 3.

Besides the enabling of the pointer aliasing for backwards predicate analysis, further ways of optimizing backwards analyses in general might be recognized by a more closely investigation. One aspect which can be examined more extensively is the impact of the used ordering of the `waitlist` (compare Algorithm 1) in backwards analysis. In this work, only the ordering according to BFS and DFS traversal are compared. These two traversal options seemed not to make a huge difference for both analysis directions and both investigated analysis. Since the ordering only influences the analysis where a specification violation is found [32], the comparison between these two options may be more precise if only the verification results '*false*' are considered. Additionally to the BFS and DFS ordering, further ordering schemes could be evaluated in the use in backwards analyses. Possible other schemes comprise the preference of deep branches, subsequent treatment all statements of a called function, or the prioritizing of states with less threads.

Moreover, other model checking approaches could be implemented for the investigation in backwards analysis. Therefore, analyses that track variables and assignments in most instances symbolically are suited, which is the case for the *SymbolicValue-Analysis*, *IntervalCPA*, *OctagonCPA*, and SMT-based analysis like *Impact*, *BMC*, or *PDR* that are already implemented in CPAchecker [17] for forwards analysis.

Last but not least, establishing a concept for the combination of forwards and backwards program analysis appears to be promising as it had been shown that the combination of backwards and forwards traversal in model checking approaches is more powerful than one analysis direction on its own [10]–[16]. This concept could be realized by starting the analysis at both locations, l_{INIT} and l_{ERR}, at once. Then the reachability analysis is started from l_{INIT} in forwards direction and form l_{ERR} in backwards direction. If the analyses meets at a location, the states have to be merged and the traversal has to be stopped for these branches. The advantage of this approach is, that potentially a smaller state space is explored by only examining the region of interest, which are paths from the entry of the program l_{INIT} to the investigated error location l_{ERR}.

References

[1] D. Beyer and T. Lemberger, "Software Verification: Testing vs. Model Checking," *Proc. HVC*, pp. 99–114, 2017.

[2] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the State Explosion Problem in Model Checking," *Lecture Notes in Computer Science*, no. January, 2001.

[3] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

[4] J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic Model Checking with Partitioned Transition Relations," *Proceedings of the 1991 International Conference on Very Large Scale Integration*, 1991.

[5] J. R. Burch, E. M. Clarke, and K. L. McMillan, "Symbolic model checking: 10ˆ20 states and beyond," *Information and Computation*, vol. 98, pp. 142–170, 1992.

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[7] D. Beyer and S. Löwe, "Explicit-state software model checking based on CEGAR and interpolation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pp. 146–162, 2013.

[8] T. Seufert and C. Scholl, "Sequential Verification Using Reverse PDR," *MBMV*, pp. 79–89, 2017.

[9] T. A. Henzinger, "From Pre-historic to Post-modern Symbolic Model Checking," *Computer Aided Verification*, pp. 195–206, 1998.

[10] T. Seufert and C. Scholl, "Combining PDR and reverse PDR for hardware model checking," *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018*, vol. January, pp. 49–54, 2018.

[11] D. Hutchison and J. C. Mitchell, "Invariant Checking Combining Forward and Backward Traversal," *Lecture Notes in Computer Science*, vol. 9, no. 3, p. 414, 1973.

[12] G. Cabodi, S. Nocco, and S. Quer, "Mixing Forward and Backward Traversals in Guided-Prioritized BDD-Based Verification," *Computer Aided Verification*, pp. 471–484, 2002.

[13] S. G. Govindaraju and D. L. Dill, "Verification by approximate forward and backward reachability," *ICCAD '98 Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pp. 366–370, 1998.

[14] P. Cousot and R. Cousot, "Refining Model Checking by Abstract Interpretation," *Automated Software Engineering*, vol. 6, no. 1, pp. 69–95, 1999.

[15] G. Cabodi, M. Murciano, S. Nocco, and S. Quer, "Stepping forward with interpolates in unbounded model checking," *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, no. May 2014, pp. 772–778, 2006.

[16] V. D'Silva, M. Purandare, and D. Kroening, "Approximation refinement for interpolation-based model checking," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4905 LNCS, pp. 68–82, 2008.

[17] D. Beyer and M. E. Keremoglu, “CPAchecker: A Tool for Configurable Software Verification,” pp. 184–190, 2011.

[18] D. Beyer and A. Stahlbauer, “BDD-based software model checking with CPAchecker,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7721 LNCS, pp. 1–11, 2013.

[19] D. Beyer, M. E. Keremoglu, and P. Wendler, “Predicate abstraction with adjustable-block encoding,” *Formal Methods in Computer Aided Design*, pp. 189–197, 2010.

[20] D. Beyer and P. Wendler, “Algorithms for Software Model Checking : Predicate Abstraction vs . I MPACT,” *Proc. FMCAD*, pp. 106–113, 2012.

[21] D. Beyer, “Automatic Verification of C and Java Programs: SV-COMP 2019,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11429 LNCS, pp. 133–155, 2019.

[22] T. A. Henzinger, R. Jhala, and R. Majumdar, “Lazy Abstraction,” *POPL*, 2002.

[23] D. Beyer, T. A. Henzinger, and G. Théoduloz, “Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis,” *Computer Aided Verification*, pp. 504–518, 2007.

[24] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, “Software model checking via large-block encoding,” *9th International Conference Formal Methods in Computer Aided Design, FMCAD 2009*, pp. 25–32, 2009.

[25] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, “The software model checker Blast,” *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[26] D. Beyer, T. A. Henzinger, and and Gregory Theoduloz, “Program analysis with dynamic precision adjustment,” *Proceedings of the 23rd International Conference on AutomatedSoftware Engineering (ASE), ACM*, pp. 29–38, 2008.

[27] P. Wendler, “Towards Practical Predicate Analysis,” *Dissertation*, 2017.

[28] D. Beyer, M. Dangl, and P. Wendler, “A Unifying View on SMT-Based Software Verification,” *Journal of Automated Reasoning*, vol. 60, no. 3, pp. 299–335, 2018.

[29] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, “Handbook of model checking,” *Handbook of Model Checking*, pp. 1–1210, 2018.

[30] D. Beyer and A. Stahlbauer, “BDD-based software verification,” *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 5, pp. 507–518, 2014.

[31] S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. Von Rhein, “Domain types: Abstract-domain selection based on variable usage,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8244 LNCS, no. May, pp. 262–278, 2013.

[32] D. Beyer and K. Friedberger, “A light-weight approach for verifying multi-threaded programs with CPAchecker,” *Electronic Proceedings in Theoretical Computer Science, EPTCS*, vol. 233, no. Memics, pp. 61–71, 2016.