University of Passau
Faculty of Computer Science and Mathematics
Chair for Software Systems

UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Master's Thesis

Block-Abstraction Memoization as an Approach to Verify Recursive Procedures

Karlheinz Friedberger

March 11, 2015

Supervisors: Prof. Dr. rer. nat. Dirk Beyer
Prof. Dr.-Ing. Sven Apel
Advisor: M. Sc. Philipp Wendler

Abstract

Block-abstraction memoization (BAM) is a technique in program verification that divides a program into blocks (like functions or loops) and analyses them separately. If a block is inspected several times, BAM uses cached results of the block's former analyses to gain performance.

In this thesis, the operators *reduce* and *expand* of BAM are formally defined in an analysis-independent way for the first time. Then BAM is extended with a fixpoint algorithm and a third operator *rebuild* such that recursive procedures can be analyzed in an abstract manner. The fixpoint algorithm aborts the recursion if every function is unrolled far enough to prove the program's safety in a sound way. The operator *rebuild* allows to handle the problem of equal identifiers in recursive function calls.

As the approach of BAM is independent from the underlying analysis, BAM is specified for the predicate analysis (that uses lazy abstraction, predicates and formulae to analyze a program) and the value analysis (that simply stores variables and their assigned values in a map). The refinements of both analyses are modified to support the verification of recursive procedures within the CEGAR approach. In case of the predicate analysis several interpolation strategies for recursive programs are compared.

The implementation in the framework CPAchecker is evaluated on a set of recursive source files and the results are competitive with other software model checkers that participated in the recent International Competition on Software Verification 2015.

Abstrakt

Block-Abstraction Memoization (BAM) ist eine Methode bei der Programmverifikation um den zu analysierenden Zustandsraum zu verringern. Dabei wird ein Programm in Blöcke zerlegt und diese einzeln analysiert. BAM verwendet einen Cache und verwendet Ergebnisse von früheren Analysen wieder, wenn derselbe Block mehrmals betrachtet wird.

Diese Arbeit definiert die Operatoren *reduce* und *expand*, welche von BAM genutzt werden, zum ersten Mal formal und unabhängig von der verwendeten Analyse. Außerdem wird BAM mit einem Fixpunktalgorithmus und einem Operator *rebuild* erweitert, damit rekursive Programme in einer abstrakten Art und Weise analysiert werden können. Der Fixpunktalgorithmus stellt fest, wann jede (rekursive) Funktion weit genug abgerollt ist um die Sicherheit des Programms bzgl. der Spezifikation zu garantieren. Der Operator *rebuild* hilft bei der Behandlung von gleichnamigen Variablen in rekursiven Funktionsaufrufen.

Weil BAM unabhängig von der verwendeten Analyse arbeitet, werden die Operatoren von BAM sowohl für die Prädikatenanalyse, welche mit Hilfe von Formeln, Prädikaten und Abstraktionsberechnungen ein Programm analysiert, als auch für die Value-Analyse, welche nur Variablen und deren zugewiesene Werte kennt, spezifiziert. Die Refinement-Prozeduren beider Analysen wurden geändert, um die Verifikation von rekursiven Programmen mit CEGAR zu ermöglichen. Des Weiteren werden mehrere Interpolationsstrategien verglichen, die von der Prädikatenanalyse bei rekursiven Programmen eingesetzt werden können.

Zuletzt folgt die Evaluierung der Implementierung im Framework CPAchecker auf einer Reihe von rekursiven Programmen. Die Ergebnisse sind konkurrenzfähig mit anderen Software-Model-Checkern, welche an der International Competition on Software Verification 2015 teilgenommen haben.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Passau, 11.03.2015, Karlheinz Friedberger

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Model Checking and Software Verification

Model checking is an algorithmic technique to verify that a system description satisfies a specification. A software implementing such algorithms is called a model checker and either proves the system description against the specification or reports a counterexample violating the specification.

Software verification is the application of model checking to implemented code (programs) and is researched since the early days of computer science. Tony Hoare describes the correctness of computer programs as "the fundamental concern of the theory of programming and of its application in large-scale software engineering" [22]. He emphasizes the importance of achieving the goal of automatic software verification not only for end-users but also for the economy and considers automatic verification of software as one of the grand challenges for computing research. Verifying software is undeniably a challenging task, because even the problem whether or not a program terminates is not decidable in general.

The basic idea of software verification is checking, if the (maybe infinite) state graph of a program is a model of its specification. Given a program description (in form of source code) and a logical specification (for example as temporal safety property), software verification returns either the proof of correctness for the program or a specific execution path through the program as counterexample. This approach is faced with the problem of representing large numbers of states for more complex systems. To handle such situations, states are often represented through abstractions with omitting unimportant parts of the analyzed program or in an efficient (maybe canonical) form such as intervals, SMT formulae or BDDs. Another method to reduce the potential number of states is to divide the program into smaller parts, verify each part separately, and then merge the results in a logical and sound way.

This thesis covers model checking of recursive program, which is more complex than the verification of non-recursive software, because of the additional problems of function calls that cause shadowing of identifiers, the explicit usage of the call stack to store assignments, and the possibly unbounded deep of a recursive function

call. We provide the theory to analyze recursive programs in an abstract manner such that we are able to verify recursive procedures without having to unroll them completely, which may be infeasible and sometimes impossible. The approach to verify recursive software is specified for the value analysis and the predicate analysis and the working implementation in the framework CPAchecker is explained and evaluated.

1.2 CPAchecker as Verification Framework

CPAchecker is the framework for configurable software verification, where the implementation of this thesis is integrated. This section gives an overview of CPAchecker, its structure, and basic components.

1.2.1 Overview

The framework CPAchecker[1] [7] is currently developed at the Software Systems Lab at the University of Passau and has many contributors, also from other organizations. In the last years CPAchecker was always one of the best model checkers in the International Competition on Software Verification[2]. Each year one or more different configurations of CPAchecker (each configuration from a distinct participant) succeed in the competitions. In 2012, the first year of the competition, the winning configuration of CPAchecker used block-abstraction memoization (BAM) with predicate analysis. Other winning configurations of CPAchecker are based on predicate analysis or value analysis, however BAM was never used in a competition after 2012.

1.2.2 Structure

The framework CPAchecker was developed with the concepts of configurable program analysis (CPA) and counterexample-guided abstraction refinement (CEGAR) in mind such that new analyses can easily be implemented in a standardized way [7]. Both concepts are formally described later in Section 2.2 and 2.3. By providing a modular and intuitive structure this concept aims to facilitate the implementation, configuration, combination, and comparison of different program analyses in a convenient and logical way. The default control flow of CPAchecker shown in Figure 1.1 generates control flow automata (CFA) from source code and runs a (parametrized)

[1] `http://cpachecker.sosy-lab.org` - last check: March 5, 2015
[2] `http://sv-comp.sosy-lab.org` - last check: March 5, 2015

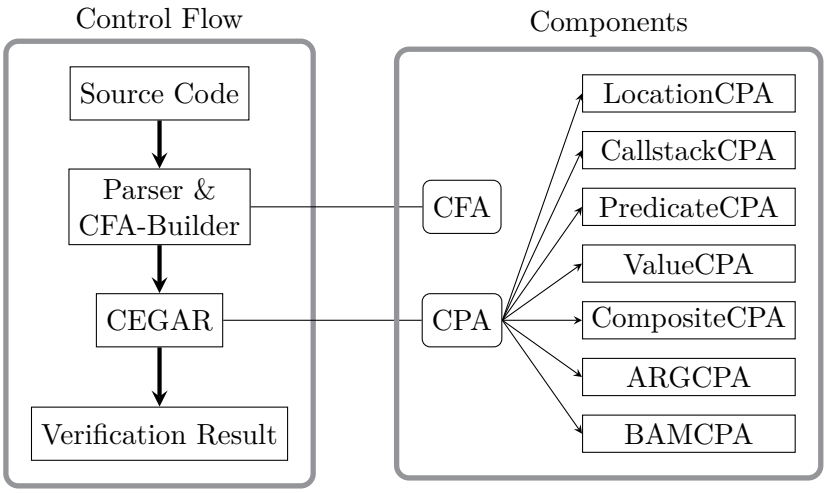


Figure 1.1: Overview of the control flow and basic components in CPAchecker

algorithm on them. This algorithm is mostly a CEGAR-Algorithm using CPA for further analysis. CPAs can be nested within each other and allow the exchange of information between several distinct analyses. CPAchecker has bindings to several external tools that can be accessed from the analyses, like for example libraries for SMT solvers, BDDs, and octagons.

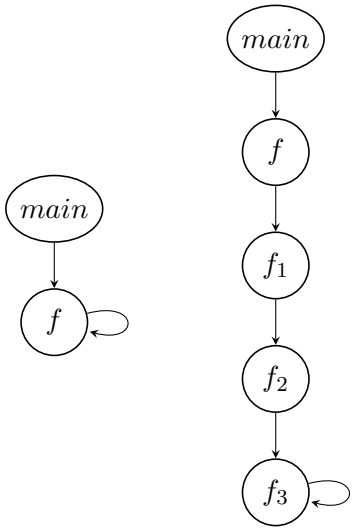
1.3 Related Work

As software verification is an important topic of research since many years, there exist several approaches to analyze recursive procedures. This section will mention some of the most important approaches for this thesis like bounded model checking and interpolation based methods.

1.3.1 Bounded Model Checking

A bounded model checker like CBMC[3] can easily handle language features like recursion, because the analysis unrolls the recursive function up to a specified maximum depth exactly in the same manner as loops, only with tracking an explicit stack for variables and functions. The main problem of bounded model checking is finding the right bound. As the verifier ignores any deeper recursion, the analysis might be unsound and miss a bug.

[3] `http://www.cprover.org/cbmc` - last check: March 5, 2015




```
1  void main() {
2    int a = 2;
3    int b = f(a);
4    if (b != 2) {
5      error();
6    }
7  }
8
9  int f(int x) {
10   if (x <= 0) {
11     return x;
12   } else {
13     int tmp = f(x - 1);
14     return tmp + 1;
15   }
16 }
```

Figure 1.2: Simple recursive program (call graph, unrolled call graph, source code)

Bounded Function Unwinding in CPAchecker

CPAchecker had no direct support for recursion in its CPA- and CEGAR-Algorithms before the implementation of this thesis. Because of the framework's modular structure, only one component (named CallstackCPA) knows about functions and the call stack. Most other components can not handle recursive procedures, because they only know the current scope and (only in case of function call or return) also the calling function. They assume that a function's name is unique in the current scope and thus a function calling itself can not be handled, because the identifiers of variables (e. g. "f::x" for a variable "x" in a function "f") would collide, when several scopes of identically named functions are valid at one program location. A collision also happens in case of transitive recursive function calls, for example, if a function f calls another function g and g calls f again.

However there exists a feature in CPAchecker that allows to copy (or clone) a function f and assign a new name (for example with an index, like f_1) for the new function. With this little trick recursive procedures can be unrolled a certain number n of steps (using the copied functions f_1, f_2,... until f_n). The number n of unrollings is static and must be given by the user, because all manipulation of the program (like changing or copying functions) has to be done before the analysis, as many components of CPAchecker rely on this.

In Figure 1.2 the function f is recursive and calls itself. Thus the call graph

contains a loop for the function f. With unrolling the call graph several times (here with $n = 3$), the recursive function call is deep in the call graph such that an analysis like the value analysis does not reach it (assuming a full precision, i. e. tracking all variables and values of the program such that the criteria for aborting the recursion can be computed).

This approach does only work with skipping the recursive function in the unwound graph or without the CEGAR-Algorithm. The reason for the first case is simple: skipping recursive functions, i. e. assuming non-deterministic assignments for all global variables, the function's parameters and the return value, is sound and no analysis must handle recursion. The second case disables CEGAR and thus causes the usage of a full precision for the analysis such that all available variables are tracked. In contrast to that, CEGAR would start its iteration with an empty precision (without tracking any variables) and in its first iteration the recursive function would be unrolled an infinite number of steps, because the termination criteria (or its variables) of the recursive function are not analyzed and the exploration visits every program path as far as possible. The current implementation in CPAchecker aborts the analysis, if a recursive function calls is reached.

1.3.2 Interpolation-Based Approach for Recursive Procedure Calls

There are already several approaches and tools that are based on interpolation to analyze recursive programs [1, 17]. As one of the best analyses implemented in CPAchecker also depends on abstraction and interpolation, here other tools are mentioned that try to handle recursion on a more abstract level than just explicitly tracking a stack for functions and variables.

Function Summaries: Whale

Whale [1] is an extension of Impact [25] and allows interprocedural program analysis. Just like Impact it is integrated within the CEGAR-approach and uses interpolation to get invariants for the refinement. Whale uses two types of formulae, namely state- and transition-interpolants, to get summaries of functions. The formulae are build through a mixture of sequential and well-scoped interpolation, which are both described in Section 2.6, combined with several formula transformations like substitutions or existential quantification of identifiers. As the implementation of Whale is only a prototype, its application to a bigger number of source files was not possible. Even a re-evaluation of the results provided in the official paper was not achievable.

Nested Interpolants: ULTIMATE

The most influential approach for (the interpolation-based part of) this thesis (i. e. the operator *rebuild* in the predicate analysis) is based on the theory of nested interpolants [17], which can be seen as predecessor of tree interpolation and is implemented in the framework ULTIMATE[4]. Instead of using a tree as intermediate data structure, the formulae are directly used as input for Craig interpolation to get nested interpolants.

The key idea of nested interpolants is to use the information that is available at the entry of a function for all interpolants inside the function's scope. After returning from the function the formulae is updated with the formulae for parameter and return value assignment. This is very similar to the approach of nested or tree interpolation implemented in CPACHECKER, which however only uses the parameter assignment for rebuilding and does not need the assignment of the return value as integral part of the interpolation procedure. This coincides better with the transfer relation of the used analysis, because we do not need an additional transfer relation for returning from a recursive function and keep every access to further data for the operator *rebuild* (that is defined in Section 3.3.3) as limited as possible.

As nested interpolation is the predecessor of tree interpolation, it also distinguishes between function calls with and without a corresponding function return such that different formulae and steps are used in the interpolation procedure for both cases.

1.3.3 Fixpoint-Algorithm: CPAREC

CPAREC[5] is a software model checker that uses an intraprocedural (recursion-free) program analyzer like CPACHECKER as underlying verifier [11]. CPAREC participated in the International Competition on Software Verification 2015 and scored slightly better than the competing version of CPACHECKER that used BAM combined with the predicate analysis for the category of recursive program files. CPAREC reached a score of 18 points of 40, CPACHECKER only 16. The tool is also evaluated in this thesis (see Section 6.4) to compare it with CPACHECKER on a bigger set of source files.

Figure 1.3 shows the main algorithm of CPAREC, which is a fixpoint algorithm. It iterates until the minimal number of unwindings for recursive function calls is determined or a valid counterexample leading to an error is found. The tool manipulates (refines) the control flow of the source file such that additional assertions are

[4] `http://ultimate.informatik.uni-freiburg.de` - last check: March 5, 2015
[5] `https://github.com/fmlab-iis/cparec` - last check: March 5, 2015

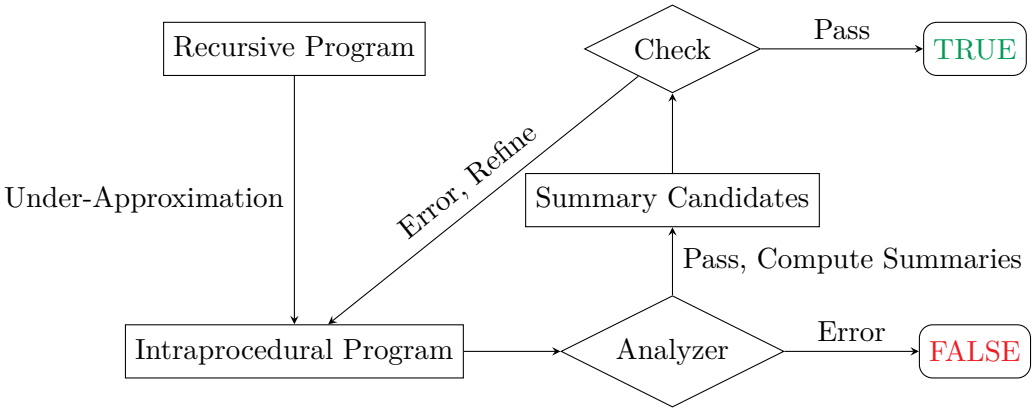


Figure 1.3: Control flow in CPArec

inserted before and after all function calls and recursive functions are either unrolled up to a specified bound or replaced by the corresponding function summary. The upper bound is incremented in each execution of the loop. To unwind a function, a copy of it is created with a renamed identifier (for example with an incremented index) and called instead of the original function. The program analyzer checks the transformed program. If an error is found, the algorithm terminates and reports the counterexample. Otherwise the output of the wrapped program analyzer is used to compute function summaries for called procedures. If all summaries are satisfied, the source file is proven as safe. Otherwise the iteration continues.

Similar to the analysis used in this thesis the tool CPArec uses a fixpoint algorithm and is (nearly) independent from the underlying analysis, as long as some kind of invariants can be extracted from the wrapped analyzer's output. According to the tool's description, the wrapped program analyzer can be switched without great effort if appropriate tool bindings are implemented.

However in contrast to CPAchecker, there is no benefit for CPArec through internal data-structures like the cache of BAM. Thus the whole (further unwound) control flow has to be analyzed again in each iteration of the fixpoint algorithm. Another interesting difference between CPAchecker and CPArec is the fact that CPAchecker uses a fixpoint algorithm for under-approximating the state space, surrounded by CEGAR, which causes an over-approximation due to a too coarse precision. The control flow of CPArec is exactly opposite, because the wrapped program analyzer may use CEGAR and the surrounding algorithm computes the fixpoint for the recursion.

2 Background

This chapter provides the theoretical background for this thesis and provides necessary information about the framework CPAchecker, its components, data structures and algorithms. A short description Floyd-Hoare logic is followed by a chapter about interpolation and properties of interpolants.

2.1 CFA - Control Flow Automaton

A program in an imperative programming language like C can be modeled through a control flow automaton (CFA). A CFA $A = (L, l_0, G)$ is defined by a set L of program locations, an initial program location $l_0 \in L$, and a set $G \subseteq L \times Ops \times L$ of possible transitions between program locations. The program language allows a set Ops of operations to perform a transition between two program locations (e. g. assignments, assumptions, or function calls). The program locations L model the program counter and l_0 represents the program entry (where the program counter is zero).

The subset $L_E \subset L$ represents target locations (error locations) that do not match the specification. A program is considered `SAFE` if no error location from L_E is reachable from the initial location l_0 via a valid sequence of transitions in the CFA. Otherwise the program is `UNSAFE`. CPAchecker support the specification of targets as automaton such that the target is independent from the CFA and compatible with further analyses. An example beyond program locations would be properties of variables or memory properties like invalid pointer dereference or memory leak.

Every function f of a program is represented as a single CFA $A_f = (L, l_0, G)$. The edges of a graph A_f match statements in the function f. The CFAs of all functions are connected with call- and return-edges, which handle the assignment of parameters and return value, into one interprocedural CFA A that corresponds the whole program.

The interprocedural CFA for the recursive program previously given in Figure 1.2 is provided in Figure 2.1. For each of the functions *main* and f, there is a separate CFA named A_{main} and A_f. The functions entry nodes are labeled with 0 and 10

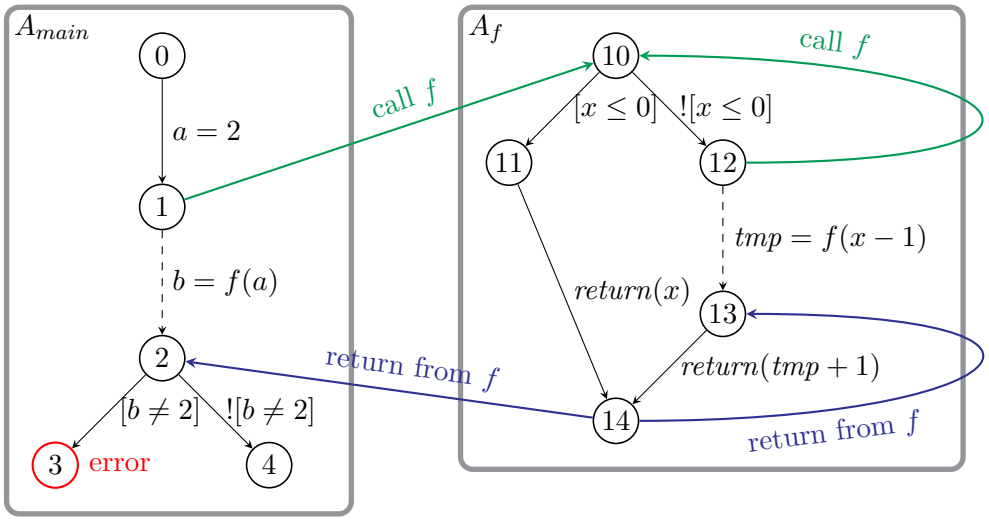


Figure 2.1: CFA for the example program in Figure 1.2 with interprocedural edges

and the exit nodes with 4 or 14, respectively. The call-edges are marked green, for return-edges the color blue is used. Due to the recursive control flow of the function f the CFA A_f has a call-edge and a return-edge to itself, i. e. the function f calls itself. Other edges contain either an assignment like $a = 2$ or an assumption in brackets like $[b \neq 2]$. The target location of the program is marked red and labeled with 3.

Figure 2.1 shows a simplified CFA, which exactly matches the program's statements and assumptions, however ignores types of variables given in the source code. The implementation in CPACHECKER uses several additional nodes and edges for temporal variables that are omitted here for readability. In this thesis CFA is always used to denote the interprocedural CFA A of the whole program.

2.2 CPA - Configurable Program Analysis

A concept for program analysis is configurable program analysis (CPA) [5], which was later extended to CPA with dynamic precision adjustment (CPA+) [6]. A CPA+ $\mathbb{D} = (D, \Pi, \rightsquigarrow, merge, stop, prec)$ is an abstract reachability analysis for a CFA $A = (L, l_0, G)$ and consists of an abstract domain D, a set Π of precisions, a transfer relation $\rightsquigarrow$, and three operators $merge$, $stop$, and $prec$. In the following E always denotes a (possibly infinite) set of elements and CPA always denotes CPA+. This section contains a description of each component of a CPA.

Abstract Domain and Abstract State The abstract domain $D = (C, \mathcal{E}, [[\cdot]])$ appoints the target of the analysis and determines, which aspects of the program should be analyzed. It is defined by a set C of concrete states, a lattice $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top, \bot)$, and a concretization function $[[\cdot]]$.

Let $\sqsubseteq \subseteq E \times E$ be a partial order over the elements of E. The least upper bound of a set $M \subseteq E$ is defined as the smallest element $e \in E$ that satisfies $e' \sqsubseteq e$ for all $e' \in M$, formally $min\{e \in E | \forall e' \in M : e' \sqsubseteq e\}$. The greatest lower bound of a set $M \subseteq E$ is defined as the biggest element $e \in E$ that satisfies $e \sqsubseteq e'$ for all $e' \in M$, formally $max\{e \in E | \forall e' \in M : e \sqsubseteq e'\}$. The join operator $\sqcup : E \times E \to E$ returns the least upper bound for its two parameters and is commutative ($e \sqcup e' = e' \sqcup e$). A lattice $(E, \sqsubseteq, \sqcup, \top, \bot)$ is defined by a partial order $\sqsubseteq \subseteq E \times E$ if there is a least upper bound for every subset $M \subseteq E$. The least upper bound of the whole set E is the top element $\top = \bigsqcup E$. The greatest lower bound of the lattice is the bottom element $\bot$. Abstract states are defined by the lattice elements E. The concretization function $[[\cdot]] : E \to 2^C$ associates an abstract state e with the set $C_e \subseteq C$ of concrete states represented by e. The set C_e may be infinite. For convenience the concretization function $[[\cdot]]$ is also defined for a set $S \subseteq E$ of abstract states with $[[\cdot]] : 2^E \to 2^C$ and $[[S]] = \bigcup_{e \in S} [[e]]$.

Precision The set Π of precisions specifies the precisions of the abstract domain. The precision determines the granularity of the program analysis and for example specifies which program variables should be tracked. The program analysis keeps track of different precisions $\pi \in \Pi$ for different abstract states $e \in E$. A tuple $(e, \pi) \in E \times \Pi$ is denoted as abstract state e with precision π. In the following the precision is omitted, if it is not necessary to describe the functionality or behavior of a component.

Transfer Relation For each abstract state $e \in E$, the transfer relation $\rightsquigarrow \subseteq E \times E \times \Pi$ returns possible new abstract states $e' \in E$ that are successors of e depending on the precision $\pi \in \Pi$. We write $e \rightsquigarrow (e', \pi)$ if $(e, e', \pi) \in \rightsquigarrow$. The transfer relation is coupled with an edge $g \in G$ in the implementation of most CPAs and then also denoted with $e \overset{g}{\rightsquigarrow} (e', \pi)$. However for BAM does not depend on this and the transfer relation can be coupled with a whole set of edges (which is called a block and described in Section 2.4.6).

Merge Operator The operator $merge : E \times E \to E$ combines two abstract states into a new abstract state. The operator weakens its second parameter depending on the first parameter, thus the operator merge is not commutative. With respect to the lattice the result of $merge(e, e')$ can be anything between e and $\top$. The operator $merge$ may be based on the join operator $\sqcup$ of the lattice. Two important merge operators are $merge_{sep}(e, e') = e$ and $merge_{join}(e, e') = e \sqcup e$.

Stop Operator The operator $stop : E \times 2^E \to \mathbb{B}$ analyses if the abstract state e (first parameter) is covered by the set R (second parameter) of abstract states such that the analysis can stop. Otherwise the analysis continues with the next abstract state. Coverage is satisfied if all concrete states represented by e are part of the concretization of one or more abstract states in R, formally $[[e]] \subseteq [[R]]$. The operator $stop$ may be based on the partial order $\sqsubseteq$ of the lattice. Two important implementations of this operator are $stop_{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$ and $stop_{join}(e, R) = (e \sqsubseteq \bigsqcup R)$.

Precision Adjustment Operator For a given abstract state and a set of abstract states (each abstract state with a precision) the precision adjustment operator $prec : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ returns a new abstract state with a precision. This operator is applied after the transfer relation $\rightsquigarrow$ and updates the abstract state and its precision.

2.2.1 CPA-Algorithm

The CPA-Algorithm (Algorithm 1) computes the set of abstract states that are reachable from an initial abstract state e_0 and operates on the two sets *reached* and *waitlist*, which contain abstract states combined with their precision. Every iteration of the while-loop removes one abstract state from *waitlist* and computes its abstract successors with the transfer relation $\rightsquigarrow$. Then the dynamic precision adjustment *prec* is applied. If the analysis finds a target state, the fixpoint algorithm aborts and returns all analyzed abstract states. This eager behavior allows a direct reaction on target states in the framework. If the fixpoint of this iteration is reached, *waitlist* is empty. The algorithm merges each abstract successor with every existing abstract state in *reached*. If the merge operator produces a new abstract state, the existing abstract state is replaced with the new abstract state in the sets *waitlist* and *reached*. At last step in the iteration the stop operator checks whether the current abstract state must be analyzed further or it is already covered by another previously computed abstract state. On termination the algorithm returns the set *reached* of analyzed states.

Algorithm 1: $CPA(\mathbb{D}, R_0, W_0)$ (from [9], Algorithm 1)

Input : a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, merge, stop, prec)$,
an initial set $R_0 \subseteq E \times \Pi$ of abstract states with their precision,
s initial set $W_0 \subseteq R_0$ of frontier abstract states with their precision
Output : a set of reachable abstract states with their precision,
a set of frontier abstract states with their precision
Variables: a set $reached \subseteq E \times \Pi$ and a set $waitlist \subseteq E \times \Pi$

```
1  reached := R_0
2  waitlist := W_0
3  while waitlist ≠ ∅ do
4  |  choose (e, π) from waitlist
5  |  waitlist := waitlist \ {(e, π)}
6  |  foreach e' with e ⇝ (e', π) do
7  |  |  (ê, π̂) = prec(e', π, reached)
8  |  |  if isTargetState(e') then
9  |  |  |  return (reached ∪ {(ê, π̂)}, waitlist ∪ {(ê, π̂)})
10 |  |  foreach (e'', π'') ∈ reached do
11 |  |  |  e_new := merge(e', e'', π̂)
12 |  |  |  if e_new ≠ e'' then
13 |  |  |  |  reached := (reached ∪ {(e_new, π̂)}) \ {(e'', π'')}
14 |  |  |  |  waitlist := (waitlist ∪ {(e_new, π̂)}) \ {(e'', π'')}
15 |  |  if ¬stop(e', {e|(e, ·) ∈ reached}, π̂) then
16 |  |  |  reached := reached ∪ {(e', π̂)}
17 |  |  |  waitlist := waitlist ∪ {(e', π̂)}
18 return (reached, ∅)
```

2.2.2 ARG - Abstract Reachability Graph

The CPA-Algorithm (Algorithm 1) only tracks reached abstract states in the analysis. However the order of exploration and the dependencies between abstract states (like transfer relation, merge and coverage) are important for further analyses to extract information like an error path from the reached abstract states. This information is stored in the abstract reachability graph (ARG) that is based on all abstract states returned by a CPA and adds directed edges between them if there exists a dependency. The ARG can be produced on the fly during an analysis, because every dependency is explicitly computed in the CPA-Algorithm, when the transfer relation, the operators merge or stop are applied. The ARG-CPA $\mathbb{D}_{ARG} = (D_{ARG}, \rightsquigarrow_{ARG}, merge_{ARG}, stop_{ARG})$ is implemented as a wrapper around another CPA $\mathbb{D}_w$ and builds the ARG during the analysis. The abstract domain D_{ARG} adds information about dependencies to the wrapped abstract states

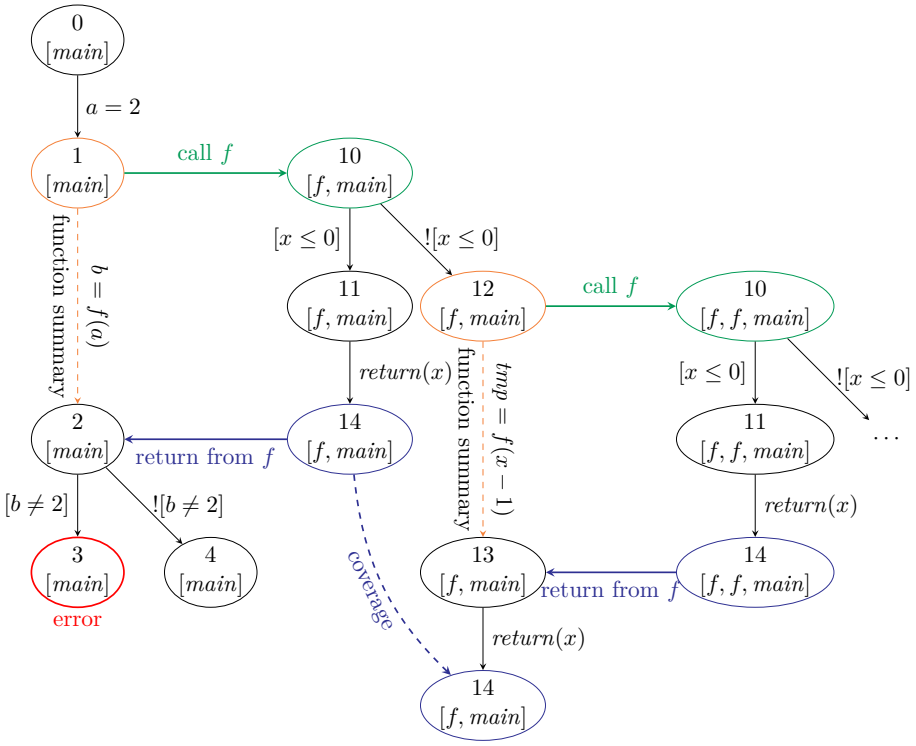


Figure 2.2: (Partial) ARG for an example program

from D_w. The operators $\rightsquigarrow_{ARG}$, $merge_{ARG}$, and $stop_{ARG}$ forward to their wrapped counterparts. The ARG-CPA does not use a precision itself, but redirect every precision-related operation to its wrapped CPA $\mathbb{D}_w$.

Figure 2.2 represents a partial ARG for the example program given in Figure 1.2. This ARG only shows the overall layout of an ARG and does not contain any information about the wrapped analysis that was used, except the program location and the call stack for each abstract state (as first and second line of each node, respectively). As edges in the ARG are based on a CFA, a labeling of the edges is provided. Call- and return-edges for function calls are marked green and blue. For better readability the function summary edges are inserted and marked orange.

There is a highlighted coverage relation between two abstract states, which depends on the used (and here not further specified) analysis. The arrow shows, which abstract state is covering another one. Abstract states are not explored further if they are covered. For these abstract states the program location and the call stack are identical, thus at least those parts of the abstract states match their counterpart. The direction of the coverage depends on the visitation strategy of the analysis, thus for each coverage relation the first found abstract state is covered by the second one.

2.3 CEGAR - Counterexample-Guided Abstraction Refinement

The precision decides the granularity of an analysis (or the level of abstraction). A stronger precision can lead to more exact results, whereas a weaker precision over-approximates the state space and might improve performance. The problem, how to choose a good precision for a program analysis, is addressed with the concept of counterexample-guided abstraction refinement (CEGAR) [14]. This is an iterative approach for an automatic refinement of the abstraction of a program analysis. Therefore the analysis has to provide a counterexample path, i. e. the ARG for the reached abstract states must be build during the analysis.

Starting with a coarse precision, CEGAR refines the precision with every found spurious target state, i. e. the improved precision prevents the found invalid counterexample from being explored again in a further iteration. The CEGAR-loop terminates if all abstract states are analyzed or a target state is confirmed (i. e. the error path is feasible). Basic elements of CEGAR are a feasibility check that verifies if a counterexample corresponds to a concrete error path, and a refinement procedure that uses an infeasible counterexample to refine the available precision and prune the reached abstract states in order to avoid finding the same counterexample in a further exploration. The information needed to refine the precision is extracted from the counterexample, for example through interpolation [1,9,25]. A detailed description of different refinements can be found in Section 4.5.2 for predicate analysis or in Section 4.4.4 for value analysis.

The CEGAR-Algorithm (Algorithm 2) starts with an initial state $e_0 \in E$ with a coarse precision $\pi_0 \in \Pi$. The algorithm returns `SAFE`, if all program states are analyzed (*waitlist* is empty) and no target state was found (in the last iteration). If a target state is found and the CPA-Algorithm returns a non-empty set *waitlist* (that contains at least one abstract state, the target state, with its precision), the error path is extracted from the set *reached*. The error path can also be a set of paths, starting in the initial program location, branching in the middle and ending in the reached target state, however this case is omitted for simplicity.

If the feasibility check confirms the counterexample, the algorithm returns `UNSAFE`. In that case the framework reports a bug and the counterexample. Otherwise the algorithm uses the infeasible error path to refine the current precision and removes all abstract states from the sets *reached* and *waitlist* that are invalidated through the refined precision.

Algorithm 2: $CEGAR(\mathbb{D}, e_0, \pi_0)$ (from [9], Algorithm 2)

Input : a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, merge, stop, prec)$,
an initial abstract state $e_0 \in E$ with a precision $\pi_0 \in \Pi$
Output : verification result *SAFE* or *UNSAFE* (with counterexample)
Variables: a set $reached \subseteq E \times \Pi$, a set $waitlist \subseteq E \times \Pi$,
an error path $\sigma = \langle l_0, ..., l_n \rangle$

```
1  reached := {(e_0, π_0)}
2  waitlist := {(e_0, π_0)}
3  while true do
4  |  (reached, waitlist) := CPA(D, reached, waitlist)
5  |  if waitlist = ∅ then
6  |  |  return SAFE
7  |  σ := extractErrorPath(reached)
8  |  if isFeasible(σ) then
9  |  |  return UNSAFE // use σ as counterexample
10 |  else
11 |  |  (reached, waitlist) := refine(reached, waitlist, σ)
```

The refinement not only removes invalid abstract states from the sets *reached* and *waitlist*, but also from the ARG. The ARG is cut off at the upper most abstract state with a new (or changed) precision. The removed subtree is re-explored with the new precision in the next iteration of CEGAR.

2.4 BAM - Block-Abstraction Memoization

Block-abstraction memoization (BAM) is a modular approach for model checking abstract state spaces. The idea of BAM is to divide a large program into smaller parts, which are called blocks and can be analyzed separately. The result of a block's analysis, i. e. the abstraction of a block, is stored in a cache in order to reuse it later. The original version of BAM [28,29] was heavily coupled with the predicate analysis. As BAM and the predicate analysis are independent and orthogonal, they are now divided from each other as far as possible in the implementation and thus also in the description in this thesis. This not only enables other analyses to be used with BAM easily, but also allows to describe BAM as an algorithm on a more abstract level. This section describes the basic structure of BAM, its operators *reduce* and *expand*, and how it is integrated in the framework CPAchecker.

2.4.1 Blocks in BAM

A basic component of BAM are blocks, which are defined as parts of a program, or more formal: A block $B = (L', G')$ of a CFA $A = (L, l_0, G)$ consists of a set $L' \subset L$ of locations and a set $G' = \{(l_1, op, l_2) \in G | l_1, l_2 \in L'\}$ of edges. A block B has a set $In(B) = \{l | l \in L' \wedge \exists (l_{prev}, op, l) \in G : l_{prev} \notin L'\}$ of input locations and a set $Out(B) = \{l | l \in L' \wedge \exists (l, op, l_{succ}) \in G : l_{succ} \notin L'\} \cup (L_E \cap L')\}$ of output locations.

In the following $\mathcal{B}$ denotes the set of all blocks and is defined as $\mathcal{B} := \{B | isBlock(B)\}$. CPAchecker supports distinct types of blocks, for instance loops or function bodies. For this thesis function bodies are used as blocks, because recursion is caused by function calls. Thus they are explicitly described in Section 3.2.2.

2.4.2 Overview of the Control Flow in BAM

BAM computes a separate set of abstract states for each block. During the analysis BAM visits the CFA and at each input location of a block a new (nested) analysis is started, which is executed until the output location of the block. The result of this nested analysis represents the block's input-output-behavior and is an abstraction of the block. The main analysis skips the whole block by using the abstraction of the block as summary. The abstraction of a block is stored in a cache. Whenever the same block is reached again under the same circumstances (i. e. with the same abstract input state and precision), BAM reuses the cached block-abstraction to improve the performance of the analysis for the whole program.

In order to make the block's abstraction more locally, two operators *reduce* and *expand* are applied before and after the analysis of a block. They remove unnecessary information from the abstract states before analyzing the block and re-add it afterwards, respectively. This makes the analysis more modular and leads to a higher cache hit rate.

2.4.3 Example for an ARG in BAM

In Figure 2.3 an example for a (simplified) ARG is given, which is produced by BAM. Each node in the ARG is labeled with an abstract state e_k with an incrementing index k, which represents the order of visitation during the analysis. For each abstract state e_k there also is a precision π_k, which however is omitted in the example. The example is based on a (not further specified) CFA with three blocks B_0, B_1, and B_2, which are partially nested within each other. The analysis performs a depth first search on the CFA. When a block is entered, it is analyzed completely before

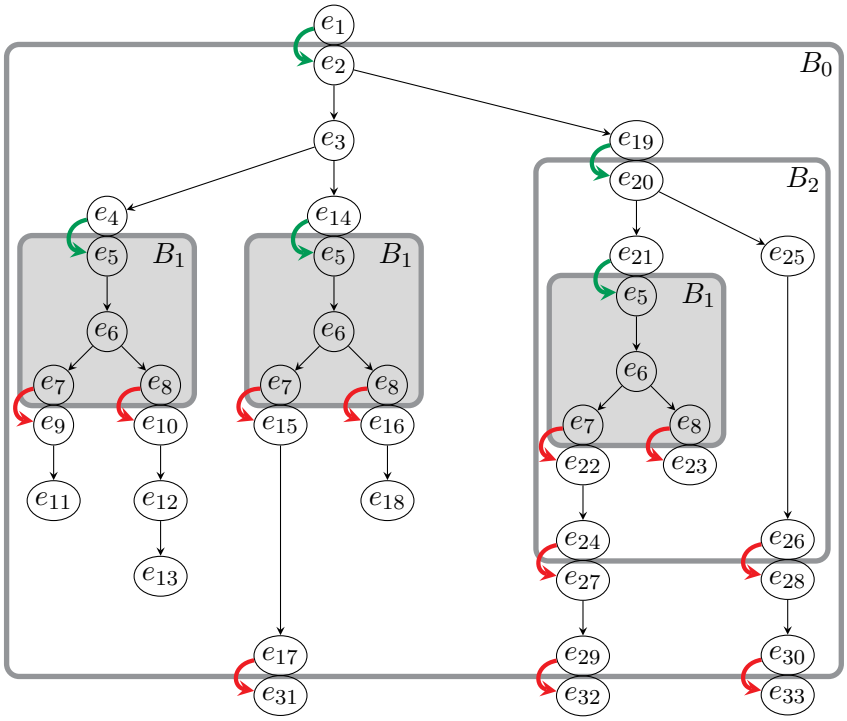


Figure 2.3: Example for an ARG with blocks produced by BAM

further exploration of the state space. Thus the ARG actually consists of separate ARGs for each block. The ARG for a block reaches from its reduced abstract input state until its abstract output states, which are expanded for further processing. The application of the operators *reduce* and *expand* is highlighted with green and red arrows, respectively.

The analysis starts with reducing the initial abstract state e_1 to e_2, which is the input state of block B_0. Then a new CPA-Algorithm is executed for program locations in block B_0, beginning with the initial abstract state e_2. A few steps later, the block B_1 is reached and the abstract state e_4 is reduced to e_5, which is the input state of block B_1. The analysis of block B_1 performs a new CPA-Algorithm with the initial abstract state e_5. The result of the algorithm consists of two sets $reached_{B_1} = \{e_5, e_6, e_7, e_8\}$ and $waitlist_{B_1} = \emptyset$. The cache uses the triple (B_1, e_5, π_5) to identify the result of the block's analysis, i. e. the sets $reached_{B_1}$ and $waitlist_{B_1}$. The abstract states e_7 and e_8 are the output states of the block B_1. After expanding the output states e_7 and e_8 to e_9 and e_{10}, respectively, the analysis of the block B_0 continues. The block B_1 is reached again with the abstract state e_{14}, which is reduced to an abstract state that matches e_5. Instead of re-analyzing the

block, the cached result of the block's former analysis is used to summarize the block. The output states of the block B_1 are expanded and the analysis of the block B_0 continues. The cached abstraction for block B_1 is also used during the analysis of block B_2, when the abstract state e_{21} is reduced to an abstract state equal to e_5.

When the analysis of block B_0 terminates, the abstract states e_{17}, e_{29}, and e_{30} are expanded and used as successors of the initial program state e_1. When the most outer CPA-Algorithm terminates in this example (without the presence of a target state), it returns the sets $reached_{main} = \{e_1, e_{31}, e_{32}, e_{33}\}$ and $waitlist_{main} = \emptyset$.

2.4.4 Cache and Memoization in BAM

BAM uses a cache that stores the input-output-behavior of each block $B \in \mathcal{B}$, i. e. the initial abstract state $e_i \in E$ (with precision $\pi_i \in \Pi$) of the block's analysis combined with the sets $reached_B \subseteq E \times \Pi$ and $waitlist_B \subseteq E \times \Pi$ that are returned after executing the CPA-Algorithm for the block B beginning with the initial abstract state e_i. The input data for the analysis of a block B is the block B itself, the initial abstract state e_i at the block entry, and its precision π_i. The triple (B, e_i, π_i) is the key for the entry in the cache, whose value is the pair $(reached, waitlist)$. The cache matches the relation $(B, e_i, \pi_i) \rightarrow (reached, waitlist)$, i. e. $\mathcal{B} \times E \times \Pi \rightarrow 2^{E \times \Pi} \times 2^{E \times \Pi}$.

The default BAM analysis uses the cache transparently to improve performance of the analysis. It does not depend on the cache, because a re-computation of cached elements is always possible, i. e. a block can always be analyzed again. However to handle recursive procedures with BAM, the cache is directly used as an important part of the analysis, because information about already analyzed parts of recursive blocks is necessary to avoid endless unrolling of a recursive procedure.

2.4.5 Reduce and Expand Operator

The analysis of BAM uses the two contrary operators $reduce : \mathcal{B} \times E \times \Pi \rightarrow E \times \Pi$ and $expand : (E \times \Pi) \times (\mathcal{B} \times E \times \Pi) \rightarrow E \times \Pi$, whose main purpose is to improve the cache-hit-rate. As shown in Figure 2.4 the initial state e_I with precision π_I of a block B is reduced with the operator $reduce$ to an abstract state e_i with precision π_i. With $reduce$ unnecessary information of an abstract state e_I and its precision π_I is removed when entering a block. The importance of some information depends on the responsible analysis. For example variables, predicates or levels of the call stack that are not accessed inside the entered block might be good candidates to be removed from the abstract state and its precision. The block B is analyzed with a new sub-analysis (a CPA-Algorithm with the transfer relation $\overset{B}{\rightsquigarrow}$) starting at the

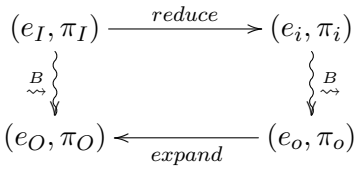


Figure 2.4: Schematic diagram of the operators *reduce* and *expand*

initial state e_i with precision π_i and returning abstract output states e_o with their precisions π_o. After the sub-analysis terminated, the operator *expand* re-adds the information to e_o and π_o that was removed with *reduce* at the block entry, to get the abstract state e_O with a precision π_O.

To guarantee the soundness of the analysis the operators *reduce* and *expand* have to fulfill the following requirement:

$$(reduce(B, e, \pi) = (e', \pi') \wedge expand(e, \pi, B, e', \pi') = (e'', \pi'')) \Rightarrow (e \sqsubseteq e'')$$

If we have a block B, reducing an abstract state e (with precision π) and expanding the result e' (with precision π') again is over-approximating the abstract state e itself. Each analysis that implements the operators *reduce* and *expand* has to fulfill this requirement, otherwise the soundness of the whole analysis is not ensured.

The default BAM analysis applies the operators *reduce* and *expand* to get a higher cache-hit-rate. BAM also works correctly (but maybe slower due to cache misses), if those operators were omitted (i. e. return identities of the given abstract states and precisions). Therefore we define two operators $reduce_{id}(B, e_I, \pi_I) := (e_I, \pi_I)$ and $expand_{id}(e_I, \pi_I, B, e_o, \pi_o) := (e_o, \pi_o)$, where the abstract states e_I and e_o represent the (not reduced) input state and the (not expanded) output state for the analysis of block B with their precisions π_I and π_o, respectively. The operators $reduce_{id}$ and $expand_{id}$ fulfill the given requirement for soundness.

For the analysis of recursive procedures it is at least necessary to reduce and expand the information about the call stack to get coverage information in order to abort unrolling recursive procedures.

2.4.6 BAM-CPA

The integration of BAM into CPAchecker is done as a CPA, similar to many other analyses. The BAM-CPA $\mathbb{D}_{BAM} = (D_{BAM}, \rightsquigarrow_{BAM}, merge_{BAM}, stop_{BAM})$ can be used in every algorithm provided by the framework, for example as CPA within the

CEGAR-Algorithm (Algorithm 2). In the following the basic components of the BAM-CPA $\mathbb{D}_{BAM}$ are described.

BAM-CPA as Wrapper-CPA

The BAM-CPA $\mathbb{D}_{BAM}$ is implemented as wrapper around another (more precise) CPA $\mathbb{D}_w = (D_w, \Pi_w, \rightsquigarrow_w, merge_w, stop_w, prec_w)$ that performs a further analysis of the program (like tracking variables or function calls). The wrapped CPA $\mathbb{D}_w$ uses the abstract domain $D_w = (C_w, \mathcal{E}_w, [[\cdot]]_w)$ defined as before by a set C_w of concrete states, a lattice $\mathcal{E}_w = (E_w, \sqsubseteq_w, \sqcup_w, \top_w)$ with a set E_w of elements and a concretization function $[[\cdot]]_w$. Additionally the wrapped CPA $\mathbb{D}_w$ needs to implement the operators $reduce_w$ and $expand_w$, which must fulfill the previously described requirement of the abstract version of *reduce* and *expand*.

If the BAM-CPA is used with the CEGAR-Algorithm (described in Section 2.3), the wrapped CPA $\mathbb{D}_w$ has to be the ARG-CPA $\mathbb{D}_{ARG}$, because CEGAR depends on tracking relations between abstract states. As the CEGAR-Algorithm is the principal (and currently only) use case of BAM in CPAchecker, there is currently no support for other wrapped CPAs $\mathbb{D}_w$ in the BAM-CPA $\mathbb{D}_{BAM}$ than the ARG-CPA $\mathbb{D}_{ARG}$. As the ARG-CPA itself is also wrapping another CPA, more complex analyses can be configured by combining several nested CPAs.

Abstract Domain and Abstract State There is no need to store any information of BAM itself in an abstract state. BAM uses the abstract domain D_w of its wrapped CPA $\mathbb{D}_w$, which is the ARG-CPA $\mathbb{D}_{ARG}$ in all relevant cases.

Merge and Stop Operator The definition of separate operators $merge_{BAM}$ and $stop_{BAM}$ for BAM is unnecessary, because they are only forwarded to their counterparts $merge_w$ and $stop_w$ in the wrapped CPA $\mathbb{D}_w$.

Transfer-Relation The most important part of BAM is the transfer relation, because most other components of BAM just forward to their wrapped counterparts. In contrast to other edge-based transfer relations, the transfer relation $\rightsquigarrow_{BAM}$ of BAM does not always only return the successors $e' \in E_w$ with $e \overset{g}{\rightsquigarrow}_{BAM} e'$ for an abstract state $e \in E_w$ such that e' is reachable from e via a single edge $g \in G$ in the CFA. The successors e' of an abstract state e at an input location $l_{in} \in In(B)$ of a block B can also be the abstract states $e' \in E_w$ with $e \overset{B}{\rightsquigarrow}_{BAM} e'$ at the output locations $l_{out} \in Out(B)$ of the block, i. e. there can be several edges (and thus multiple

ramified and merging paths) between the abstract state e at l_{in} and its successor e' at l_{out}.

The algorithms of BAM use some global variables that are explained here:

- The cache $Cache \subseteq \mathcal{B} \times E \times \Pi \to 2^E \times 2^E$ denotes the central data structure of BAM and is empty before the analysis starts.
- The block $B_{cur} \in \mathcal{B}$ represents the block of the current analysis. Whenever a new sub-analysis for a block starts or terminates, this variable is updated. The initial value of B_{cur} is $Null$.

Depending on the current program location, the transfer relation chooses between three possibilities that can be seen in Algorithm 3.

Algorithm 3: $\rightsquigarrow_{BAM}(e, \pi)$

Input : an abstract state e with a precision π
Output : succeeding abstract states of e
Variables: a program location $l \in L$

```
1 l := location(e)
2 if B_cur ≠ Null ∧ l ∈ Out(B_cur) then
      // output location: leave block B_cur
3     return ∅
4 else if ∃B ∈ B : l ∈ In(B) then
      // input location: enter block B
5     return analyseBlock(B, e, π, l)
6 else
      // forward to wrapped transfer relation
7     return {e' | e ⇝_w (e', π)}
```

Algorithm 4: $analyseBlock(B, e_I, \pi_I)$

Input : a block B that should be analyzed,
an abstract state e_I with a precision π_I
Output : abstract successors of the abstract states of e_I
Variables: two sets $reducedResult \subseteq E \times \Pi$ and $blockResult \subseteq E \times \Pi$

```
1 (e_i, π_i) := reduce(B, e_I, π_I)
2 reducedResult := getReducedResult(B, e_i, π_i)
3 blockResult := {expand(e_I, π_I, B, e_o, π_o) | (e_o, π_o) ∈ reducedResult}
4 return blockResult
```

Algorithm 5: $getReducedResult(B, e_i, \pi_i)$

Input : a block B that should be analyzed,
a reduced abstract state e_i with a reduced precision π_i
Output : result abstract states of e
Variables: four sets R_0, W_0, $targetStates$, $blockResult$ (each of them $\subseteq E \times \Pi$)

```
1  if Cache contains (B, e_i, π_i) then
2  |  (R_0, W_0) := Cache(B, e_i, π_i)
3  else
4  |  R_0 := {(e_i, π_i)}, W_0 := {(e_i, π_i)}
5  (reached, waitlist) := CPA(D_BAM, R_0, W_0)
6  Cache(B, e_i, π_i) := (reached, waitlist)
7  targetStates := {(e_t, π_t) ∈ reached | isTarget(e_t)}
8  if targetStates = ∅ then
9  |  blockResult := {(e_o, π_o) ∈ reached | location(e_o) ∈ Out(B)}
10 else
11 |  blockResult := targetStates
12 return blockResult
```

If the current program location is an output location $l_{out} \in Out(B_{cur})$ of the current block B_{cur}, there are no succeeding abstract states (in the current block) and an empty set $\emptyset$ is returned by the transfer relation. The reason for this behavior is that, if the current CPA-Algorithm terminates because of an empty *waitlist*, i. e. no more successors are found, the abstract states e_o at the output locations l_{out} are filtered from the current set *reached* and used by the wrapping analysis as basis for the successors of the current block B_{cur}. They will be expanded and used as output states of the current block in the next outer block.

If the current program location is an input location $l_{in} \in In(B)$ of a block B, the transfer relation enters the block B and computes its abstraction, i. e. the abstract states at the output locations of the block. As shown in Algorithm 4, the operator *reduce* is applied to the current abstract state e and π get a new abstract state e_i and its precision π_i. Algorithm 5 tries to get a result for the key (B, e_i, π_i) from the cache. In case of a cache miss the abstract state e_i and the precision π_i are used as basis for the new sets *reached* and *waitlist*. In case of a cache hit, the sets *reached* and *waitlist* are taken from the cache. The CPA-Algorithm 1 is executed with the sets *reached* and *waitlist*, and uses the same BAM-CPA $\mathbb{D}_{BAM}$ again to perform its analysis. After the algorithm has terminated, a cache update is performed: For the key (B, e_i, π_i) the value $(reached, waitlist)$ is stored in the cache. If target states are found during the analysis of the block, they are extracted from the resulting

set *reached* and used as abstract states e_o for further processing. Otherwise the set *reached* is filtered for abstract states e_o (with their precisions π_o) at program locations $l_{out} \in Out(B)$. In Algorithm 4 the operator *expand* is applied on the abstract states e_o with precision π_o to get the result of the block, i. e. abstract states e_O with precision π_O at the program location $l_{out} \in Out(B)$. The resulting abstract states e_O are returned as successors of e.

If the current program location is neither an input location l_{in} nor an output location l_{out} of a block, the transfer relation $\rightsquigarrow_{BAM}$ in Algorithm 3 just forwards to the wrapped transfer relation $\rightsquigarrow_w$ and returns its result.

2.4.7 BAM with CEGAR

BAM was developed to work with the CEGAR-Algorithm 2. Thus BAM has to provide methods to get and analyze a counterexample, and to update abstract states and the set *reached* during the refinement procedure. The default refinement has to handle one set *reached* and one set *waitlist* for the whole analysis. With BAM there is at least one set $reached_B$ and one set $waitlist_B$ for every analyzed block B. If a block is analyzed several times with different initial abstract states or precisions, there exist also several distinct sets for this block.

BAM abstracts from these details and provides a "normal" counterexample and the corresponding sets $reached_{view}$ and $waitlist_{view}$ for CEGAR. The implementation is hidden behind the default interface of the expected data structure and forwards every (important) modification of the counterexample and the sets $reached_{view}$ and $waitlist_{view}$ towards the corresponding sets of the blocks' analyses that are updated accordingly. The refinement of the sets $reached_{view}$ and $waitlist_{view}$ is executed as a lazy refinement that changes only the necessary parts of the corresponding sets of the blocks' analyses and keeps as much as possible unchanged for further cache accesses. The cache might also be updated during the refinement.

2.5 Floyd-Hoare Logic

Floyd-Hoare logic [20, 21] is a formal system that consists of a set of logical rules (Hoare's rules) for proving the correctness of computer programs. This section provides basic information about Hoare's rules, because these rules are used to prove the partial correctness of handling recursive procedures with BAM. Showing total correctness would require to prove the termination of the analyzed program, which is not necessary as our analysis checks only for reachability of abstract target states.

2.5.1 Hoare-Triple

A Hoare-triple $\{P\}T\{Q\}$, consisting of a program statement T and two propositional formulae P and Q, indicates that, if P is satisfied before the execution of T and T terminates, then Q is *true* after T completes. P and Q are called the pre- and the post-condition of T. A statement T can match a single program statement like an assignment or assumption. It also can correspond to a sequence of program statements, to a tree or DAG of program statements or to a whole function body of the program. In those cases the pre- and postconditions are valid before and after the whole structure, respectively.

2.5.2 Hoare's Rules

Only three of Hoare's proof rules are needed in this thesis. These include the rules of consequence, for function instantiation and for recursion. Other rules concerning for example composition of program statements, loops or branching are not considered here. Also criteria for termination will not be analyzed with Floyd-Hoare logic.

Consequence

The rule of consequence allows to strengthen the precondition P and to weaken the postcondition Q of a statement T.

$$\frac{P' \Rightarrow P \qquad \{P\}T\{Q\} \qquad Q \Rightarrow Q'}{\{P'\}T\{Q'\}} \text{ CONSEQUENCE}$$

Function Instantiation

The rule of function instantiation consists of a function f with argument a, parameter p, function assignment b, return value r and a function body B_f. The function call of f with pre- and postconditions P and Q depends on the function body with assignments of parameter $p = a$ and return value $b = r$. The parameter p is used read-only during the execution of the function body and thus has the same value after the execution. Instead of only two identifiers p and r there can also be multiple parameters and return values. They are omitted here for simplicity, however the formulae would contain terms for each of them.

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

Recursion

The rule of recursion indicates that, if the body of a function f satisfies P and Q (including the assignments of parameters and return values) under the condition that all recursive calls to the function f satisfy P and Q, then the whole function f satisfies P and Q. Because of the recursive call of f equal identifiers exist from the calling and called function f and cause collisions. For simplicity we assume an implicit renaming and omit collisions of identifiers in the proof. This is only possible, because the renaming (or an equivalent operation) can be shifted into a different part of the analysis and is handled there in a sound way.

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

2.6 Interpolation Strategies

Interpolation denotes a special type of boolean formulae (called interpolants) retrieved as intermediate part between other boolean formulae, whose conjunction is unsatisfiable. The basic idea of interpolation is to get only the necessary information from one part of the formulae and build an interpolant from it such that the conjunction of the interpolant with the other part of the formulae is unsatisfiable. Depending on the type of interpolation additional properties must be fulfilled.

Significant research was done to find algorithms for computing interpolants for various theories in SMT solving, such as integer, real number, bit-vector or even array theory and also theories with linear arithmetic, quantifier-free logic or uninterpreted functions [10,13,23]. Interpolation of boolean formulae is not deterministic in general, i. e. there can be several distinct interpolants for the same given formulae. The choice of interpolants depends on the implementation of the SMT solver and the interpolating procedure. In general there are two steps necessary to get an interpolant: First the proof for an unsatisfiable set of formulae is computed by the SMT solver and then the interpolant is extracted from the proof. It is possible to receive several interpolants from the same proof for different partitions of formulae. In the following we always assume that the solver has already computed the proof for the unsatisfiable set of formulae.

In this section several distinct interpolation strategies are explained. The strategies differ in the alignment of formulae and the computation of interpolants. The actual implementation to compute interpolants from proofs will not be shown here, but can be looked up in articles on interpolation and proofs from SMT solvers [13, 24].

First Craig interpolants are described, then a generalization on sequential interpolants and tree interpolants follows. For each of them the most important properties are mentioned. Their importance and usage for the program analysis (and especially predicate analysis) is shown in a later section.

2.6.1 Craig Interpolation or Binary Interpolation

The simplest (and most known) definition of interpolation uses only two boolean formulae φ^+ and φ^- such that $\varphi^+ \wedge \varphi^-$ is not satisfiable. For such formulae φ^+ and φ^- there always exists a boolean formula (called a Craig interpolant) $\psi = ITP_{Craig}(\varphi^+, \varphi^-)$, which consists of only symbols common to φ^+ and φ^- and contains the necessary information from φ^+ such that it contradicts φ^-. A Craig interpolant is formally defined with three properties:

- the implication $\varphi^+ \Rightarrow \psi$ holds,
- the conjunction $\psi \wedge \varphi^-$ is unsatisfiable, and
- the symbols of the interpolant ψ appear in φ^+ and in φ^-.

$$\varphi^+ \xrightarrow[\psi]{} \varphi^-$$

Figure 2.5: Craig interpolant

2.6.2 Sequential Interpolation

Binary interpolation is often insufficient for applications and thus there is an extension to sequential interpolation, where several formulae are given as input and the interpolation procedure returns a sequence of interpolants with additional properties. Figure 2.5 and Figure 2.6 show the difference in the layout of binary and sequential interpolation. Given a sequence φ_1, ... φ_n of n boolean formulae with an unsatisfiable conjunction $\bigwedge_{i=1}^{n} \varphi_i$, there exists a sequence ψ_1, ψ_2, ... ψ_{n-1} of interpolants such that

- the implications $\varphi_1 \Rightarrow \psi_1$, $\psi_{k-1} \wedge \varphi_k \Rightarrow \psi_k$ and $\psi_{n-1} \wedge \varphi_n \Rightarrow \bot$ hold, and
- the symbols of the interpolant ψ_k are common to $\psi_{k-1} \wedge \varphi_k$ and $\bigwedge_{i=k+1}^{n} \varphi_i$.

$$\varphi_1 \xrightarrow[\psi_1]{} \varphi_2 \xrightarrow[\psi_2]{} \cdots \xrightarrow[\psi_{n-2}]{} \varphi_{n-1} \xrightarrow[\psi_{n-1}]{} \varphi_n$$

Figure 2.6: Sequence of formulae with interpolants

Based on the two rules we can conclude further assumptions that are more similar to those known from binary interpolants. However these additional assumptions alone are not sufficient to produce an inductive sequence of interpolants. In the following the additional assumptions are given:

- $\bigwedge_{i=0}^{k} \varphi_i \Rightarrow \psi_k$
- the conjunction $\psi_k \wedge \bigwedge_{i=k+1}^{n} \varphi_i$ is unsatisfiable
- the symbols of the interpolant ψ_k are common to $\bigwedge_{i=0}^{k} \varphi_i$ and $\bigwedge_{i=k+1}^{n} \varphi_i$

Most SMT solvers with support for interpolation allow to generate such an inductive sequence of interpolants through their API or through receiving several binary interpolants for distinct partitions of formulae from the same proof, i. e. computing all interpolants after the same (unsatisfiable) SAT check. As the second case depends on the internal implementation of the SMT solver and interpolation is not deterministic in general, there is no guarantee for an inductive sequence of interpolants through repeated binary interpolation in an arbitrary logic of the SMT solver. However this seems to work for most SMT solvers and the most used theories. Thus it is the current default implementation in CPAchecker and for example successfully applied in the predicate analysis, which is described in Section 4.5.

2.6.3 Tree Interpolation

In computer science a "tree" is a coherent hierarchical graph, where the root node has no parent node and each other node has exactly one parent node. The concept of interpolating a sequence of formulae can be extended to tree interpolation, where each node of the tree contains a boolean formula of the interpolation problem. Then sequential interpolation is a special case of tree interpolation with a degenerated tree (see Figure 2.7) that has only one path, i. e. there is at most one child for each node in the tree, and the root node of the tree corresponds to the last formula of the sequence.

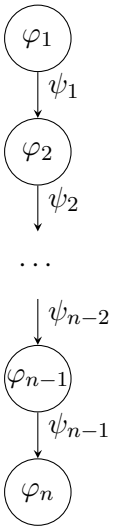


Figure 2.7: Sequential interpolation problem of Figure 2.6 as degenerated tree, each node v_i contains its labeling $L(v_i)$ and underneath $I(v_i)$ (except v_n, which is the root of the tree with labeling $I(v_n) = \bot$)

Tree interpolation is applied for verification of programs with more complex control flow like recursive [17] or multi-threaded programs [15]. Here only the description and properties of tree interpolants are given, their importance and usage for program analysis is shown in a later chapter. Especially the predicate analysis with BAM needs tree interpolation to analyze recursive procedures in a sound way (more details follow in Section 4.5.4).

The input for a tree interpolation problem is defined with a tree $T = (V, E)$, consisting of a set V of nodes and a set $E \subseteq V \times V$ of edges pointing from child to parent nodes, and a labeling $L : V \to Formula$ that assigns a boolean formula φ_v to each node v of the tree. The set of nodes in the subtree of a root-node v is denoted with $st(v) \subseteq V$ and includes the node v. Given a tree $T = (V, E)$ with an unsatisfiable conjunction $\bigwedge_{v \in V} L(v)$, there exists a labeling $I : V \to Formula$ such that

- $I(v_r) = \bot$, where v_r is the root of T,
- the implication $\bigwedge_{(w,v) \in E} I(w) \wedge L(v) \Rightarrow I(v)$ holds, and
- the symbols of an interpolant $I(v)$ are common to $\bigwedge_{(w,v) \in E} I(w) \wedge L(v)$ and $\bigwedge_{u \notin st(v)} L(u)$.

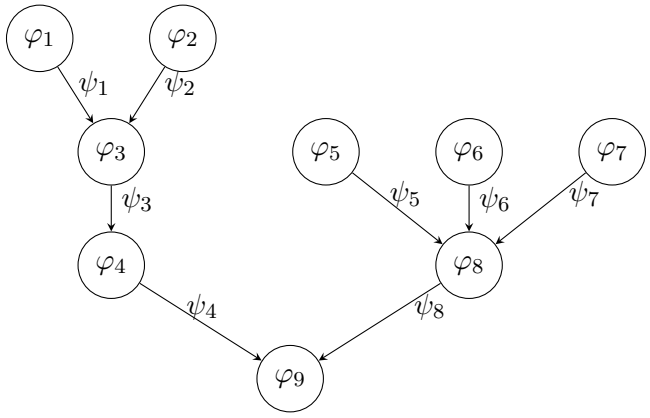


Figure 2.8: Small tree of formulae with interpolants with only 9 nodes as example, each node v_i contains its labeling $L(v_i)$ and underneath $I(v_i)$ (except v_9, which is the root of the tree with labeling $I(v_9) = \bot$)

Based on these basic rules we can conclude further rules that are weaker, but more similar to those known from binary and sequential interpolation:

- the implication $\bigwedge_{v \in st(v)} L(v) \Rightarrow I(v)$ is valid, i. e. the interpolant $I(v)$ is implied by the subtree rooted at node v,
- the formula $I(v) \wedge \bigwedge_{u \notin st(v)} L(u)$ is not satisfied, and
- the symbols of the interpolant $I(v)$ are common to $\bigwedge_{w \in st(v)} L(w)$ and $\bigwedge_{u \notin st(v)} L(u)$, i. e. occur inside and outside of the subtree rooted at node v.

Figure 2.8 shows a small example tree with only a few nodes, which are labelled with formulae φ_i and corresponding interpolants ψ_i underneath them. The definition of tree interpolants works for trees with arbitrary branching, but later only binary trees are used, i. e. each node has at most two children. They are sufficient to analyze programs with (recursive) function calls.

In the following two procedures for tree interpolation are explained to directly receive tree interpolants for an unsatisfiable set of formulae with every SMT solver supporting Craig interpolation. First well-scoped interpolation is described, which depends on the solver's internals. Then an algorithm is provided that guarantees valid tree interpolants, but has some computational overhead due to a larger number of SAT queries.

Well-Scoped Interpolation

Well-scoped interpolation [18] was one of the first approaches to handle procedures in interpolation. The idea of well-scoped interpolation is based on choosing several distinct partitionings for the formulae to generate distinct interpolants. This interpolation strategy can be defined as a tree interpolation problem, because the structure for choosing the partitions matches a tree. The tree corresponds to the analyzed program's control flow (i. e. function calls) such that all symbols (identifiers) of an interpolant are in the correct scope at the interpolation point. The benefit of this interpolation strategy is that only one proof (and thus only one query for satisfiability) is needed to generate all interpolants. As this strategy depends on the internal implementation of the SMT solver's proof, the interpolants received with well-scoped interpolation may not fulfill all requirements on tree interpolation, because there is no guarantee to retrieve valid tree interpolants for an arbitrary theory. Until now the current implementation of well-scoped interpolation in CPACHECKER did not produce any invalid tree interpolant.

The strategy to get well-scoped interpolants from a tree of formulae is given in Algorithm 6. First all formulae are asserted in the SMT solver to get a proof P for unsatisfiability. Then this (constant) proof P is used to get Craig interpolants $ITP^P_{Craig}(\varphi^+, \varphi^-)$ for all pairs of formulae φ^+ and φ^-, where φ^+ is the conjunction of all formulae of the subtree $st(v)$ for a node $v \in V$ and φ^- is the conjunction of the rest of the tree.

Algorithm 6: $ITP_{well-scoped}(T, v_r, L)$

Input : a tree $T = (V, E)$ with a root-node $v_r \in V$,
where V denotes the set of nodes and E denotes the set of edges
Output : a labeling $I : V \to Formula$
that assigns tree interpolants to all nodes of the tree
Variables: the (constant) proof P of a SAT query for an infeasible set of formulae

1 $P := checkSAT(st(v_r))$
2 **foreach** $v \in V$ **do**
3 $\quad \varphi^+ := \bigwedge_{w \in st(v)} L(w)$
4 $\quad \varphi^- := \bigwedge_{w \notin st(v)} L(w)$
5 $\quad I(v) := ITP^P_{Craig}(\varphi^+, \varphi^-)$
6 **return** I

Tree Interpolation through Binary Interpolation

Tree interpolants can be computes with binary interpolation and several distinct solver-queries. This causes some overhead for computation depending on the SMT solver's implementation, because multiple solver-queries are more expensive than direct tree interpolation, but with this approach tree interpolants can be computed with all SMT solvers that support binary interpolation. An algorithm to compute tree interpolants is given by Blanc et al [10]. There is a similar algorithm for nested interpolants [17] that is especially written for recursive procedures and thus directly refers to function calls and -returns in its description. As this is the theoretical description, program-analysis-specific parts are dismissed here and mentioned in a later section. Here the basic idea of the algorithm for general tree interpolation is described: Algorithm 7 computes interpolants from the leaf-nodes towards the root-node of the tree and uses already computed interpolants for further proceeding. In contrast to Algorithm6 for well-scoped interpolants, Algorithm 7 re-uses already computed interpolants and thus needs several distinct (unsatisfiable) SAT queries to get the needed proofs. The number of queries coincides with the number of tree nodes.

The algorithm implemented in CPAchecker is based on the version described in Algorithm 7, but differs in some details to be more efficient and to benefit from available data-structures for the tree. The loop (line 3 in Algorithm 7) can be replaced by a post-order-visitation of the tree such that all children c of a node $v \in V$ are already done, when the interpolant of the node v itself is computed. As further optimization the recursive function $S(V', v)$ in Algorithm 8 is replaced through iterations over (parts of) the post-order-sorted tree and a stack of already computed interpolants.

2.6.4 Interpolation in SMTLIB Version 2 and SMT solvers

As interpolants can be retrieved from a proof after checking satisfiability of formulae [24], interpolation is supported directly by several SMT solvers, for example MathSat[1], Princess[2], SMTInterpol[3] and Z3[4]. Table 2.1 shows the available interpolation strategies for those SMT solvers. All four SMT solvers are integrated in CPAchecker and the supported strategies are usable via a common API.

[1] `http://mathsat.fbk.eu` - last check: March 5, 2015
[2] `http://www.philipp.ruemmer.org/princess.shtml` - last check: March 5, 2015
[3] `http://ultimate.informatik.uni-freiburg.de/smtinterpol` - last check: March 5, 2015
[4] `http://z3.codeplex.com` - last check: March 5, 2015

Algorithm 7: $ITP_{treeCraig}(T, v_r, L)$ (from [10], Algorithm 1)

Input : a tree $T = (V, E)$ with a root-node $v_r \in V$,
where V denotes the set of nodes and E denotes the set of edges
Output : a labeling $I : V \to Formula$
that assigns tree interpolants to all nodes of the tree
Variables: the proof P of a SAT query for an infeasible set of formulae

foreach $v \in V$ **do**
  $I(v) := \infty$
foreach $v \in V$ *with* $I(v) = \infty$ *and* $\forall (c, v) \in E : I(c) \neq \infty$ **do**
  $\varphi^+ := S(st(v), v)$
  $\varphi^- := S(V \setminus st(v), v_r)$
  $P := checkSAT(\varphi^+ \wedge \varphi^-)$
  $I(v) := ITP^P_{Craig}(\varphi^+, \varphi^-)$
return I

Algorithm 8: $S(V', v)$ (from [10], Algorithm 2)

Input : a set $V' \subseteq V$ of nodes and a node $v \in V'$
(the tree $T = (V, E)$ and the labelings L and I are also used)
Output : the interpolant of v or the conjunction of the label of v and its children

return $\begin{cases} I(v), & \text{if } I(v) \neq \infty \\ \bigwedge_{(w,v) \in E, w \in V'} S(V', w) \wedge L(v), & \text{otherwise} \end{cases}$

SMT solver	binary	sequential	tree
MathSat	✓		
Princess	✓	✓	
SMTInterpol	✓	✓	✓
Z3	✓	✓	✓

Table 2.1: Interpolation strategies supported by SMT solvers in CPAchecker

MathSat is one of the mature solvers, however it only supports binary interpolation directly and the implementation of other interpolation strategies is based on this. There exists a proposal from Christ and Hoenicke [12] to add a new set of commands to SMT LIB version 2 [3], which is a standardized input format for SMT solvers. The commands include queries for interpolation and receiving the types of interpolation that are supported by the solver like binary, sequential or tree interpolation. Due to the importance of tree interpolation for several use cases, some solvers (for example SMTInterpol and Z3) already support it directly through their API or even on command line. The implementations in SMTInterpol and Z3 map each given binary or sequential interpolation problem onto a tree interpolation problem before solving. These two tools supports tree interpolation in all theories implemented in the solver.

The interpolation procedure in Z3 has still some restrictions that however do not affect the theory of this thesis and the implementation in CPAchecker. For example each nullary function symbol in Z3 may only occur along one path from the root to a leaf-node and all non-nullary function symbols are considered as global. Some analyses in CPAchecker (for example predicate analysis) encode program variables as nullary function symbols and uninterpreted functions as non-nullary function symbols.

3 Analyzing Recursive Procedures with Block-Abstraction Memoization

The basic analysis of this thesis is an extension of BAM that is able to analyze programs with recursive procedures. Similar to BAM, the extension also wraps another more precise analysis (like value or predicate analysis) and can be used within the CEGAR approach. This chapter starts with a recursive example program that is analyzed with BAM combined with the value analysis. Then the analysis of recursive functions as extension of BAM is explained with the fixpoint algorithm and the operator *rebuild*. At last the implementation in the CPAchecker framework is described and the soundness of the added fixpoint algorithm is proven, i. e. why no valid target state is missed in an abstract analysis.

3.1 Motivating Example

This section provides an introducing example in Figure 3.1 to get an overview of the problem of recursion and its solution through the fixpoint algorithm in BAM. Due to its simplicity the value analysis is chosen as underlying analysis, however CEGAR is omitted in favor of using a full precision, which tracks all available variables immediately. The example program contains a non-deterministically initialized variable a, which is used as input for the recursive function f. The function f must be unrolled twice to reach the target location represented by a call of the function *error*().

The CFA for the example program is given in Figure 3.2. Figures 3.3 and 3.4 show the development of the (simplified) ARGs for two iterations of the later described fixpoint algorithm (Algorithm 10). The label of a node in the ARGs consists of three parts: the first line contains two numbers that are the unique id and the program location (number of the CFA-node) of the corresponding abstract state, the second line represents the current call stack and followed by a list of all assignments available in the value analysis. If a variable is assigned $\top$, the variable is in the current scope, however its value is unknown. A special variable *ret* is used for the assignment of the return value at function returns.

```
1  void main() {
2    int a = nondet();
3    int b = f(a);
4    if (b == 1) {
5      error();
6    }
7  }
8
9  int f(int x) {
10   if (x <= 0) {
11     return 0;
12   } else {
13     int tmp = f(x - 1);
14     return tmp + 1;
15   }
16 }
```

Figure 3.1: Example program with recursion

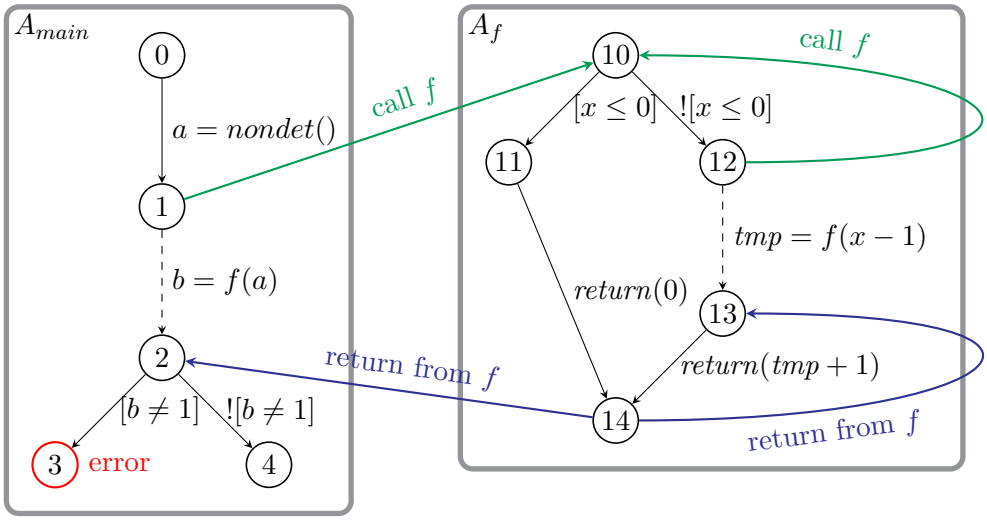


Figure 3.2: CFA for the example program in Figure 3.1

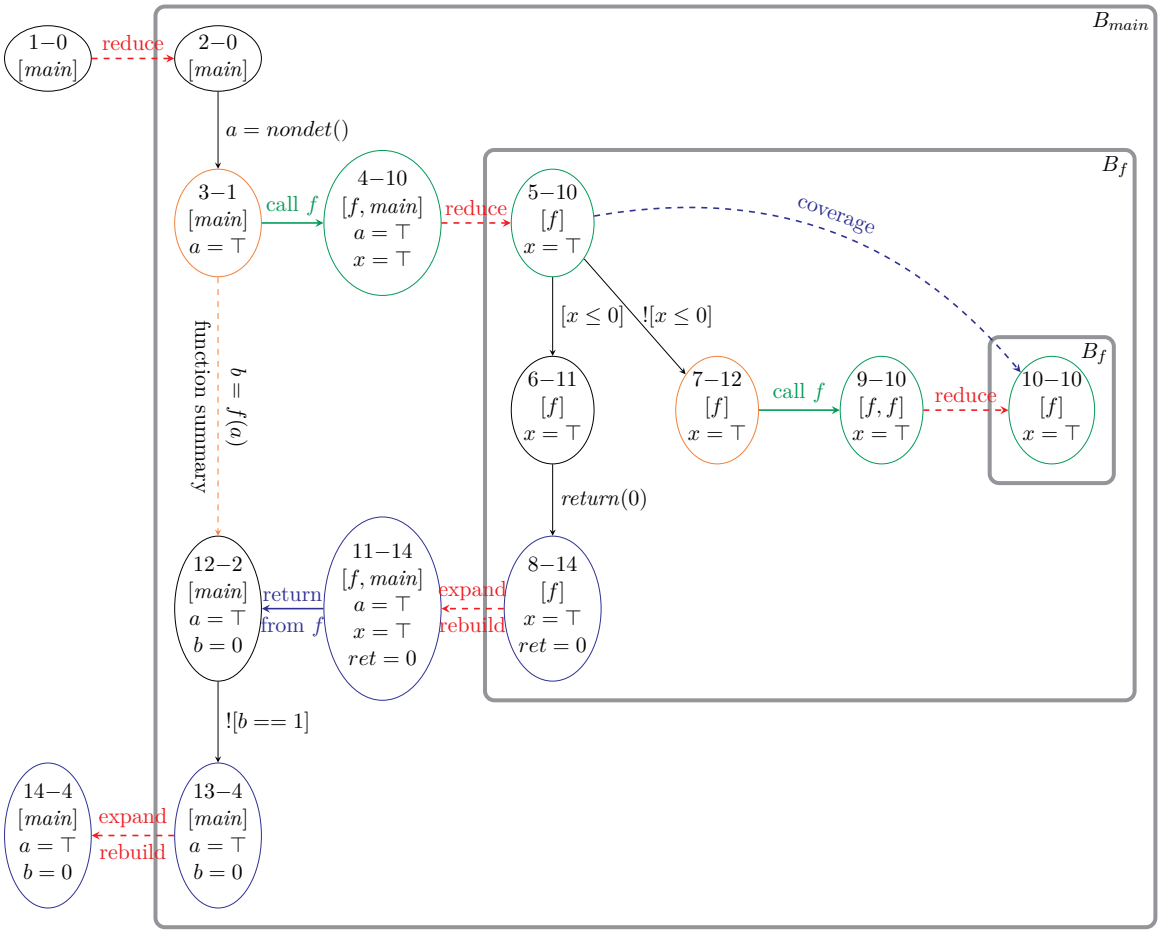


Figure 3.3: ARG produced by the value analysis (first iteration of fixpoint algorithm), each node contains its id and program location in the first line, followed by the call stack and available assignments

The effects of the operators *reduce* and *expand* are visible for abstract states at the program locations 10, 14, and 4, which are the input and output locations of the function blocks. The operator *reduce* causes the removing of all function scopes except the current one from the call stack and the assignments are also filtered for only the local variables (in the example this is only the symbol x). Those actions are reverted by the operators *expand* and *rebuild*, when the blocks are left.

In the first iteration of the fixpoint algorithm BAM explores the program's state space along the edges of the CFA, starting from the initial program location $l_0 \in L$ and entering the main block B_{main}. When the block B_f is entered the second time (with the abstract state with id 10), i. e. the function f would be unrolled again as a recursive procedure, the fixpoint algorithm notices that the reduced initial abstract state is covered by the abstract state from the first function call and tries to get an already computed result from the cache for reduced initial abstract state. As the analysis has just started, the cache is empty and thus the unrolling is aborted. The second branch of the function f is not recursive and can be analyzed further. The path via the program locations 11, 14, 2, and 4 leads to the program's exit without reaching a target location.

Now the fixpoint algorithm starts to update the abstract states in the sets $reached_i$ and $waitlist_i$ such that the program location of the recursive function call (here labeled with number 10) is re-explored in the next iteration. The second iteration benefits from the cache and the update through the fixpoint algorithm, because only the abstract states at input and output locations of blocks must be re-computed. In the example this includes all abstract states with the ids 1, 4, and 9 at the program locations 0 and 10. Several abstract states remain untouched and are not re-explored in the current analysis. In Figure 3.4 those abstract states are marked gray.

When the fixpoint algorithm reaches the recursive function call with the coverage relation in this iteration, a cache entry for the reduced initial abstract state is found and can be used to summarize the function's execution. The abstract states of the summary are an under-approximation of the function exit's state space, because only a subset of all possible abstract states is available as cached result of the block's analysis as only the non-recursive part of the block was analyzed in the first iteration of the fixpoint algorithm.

When BAM leaves a function block in further computation, it compares the current set of abstract states with a perhaps previously computed and cached result of the block's analysis. In this example, this would be done when leaving the outer block B_f at program location 14. If the comparison shows that there exist new ab-

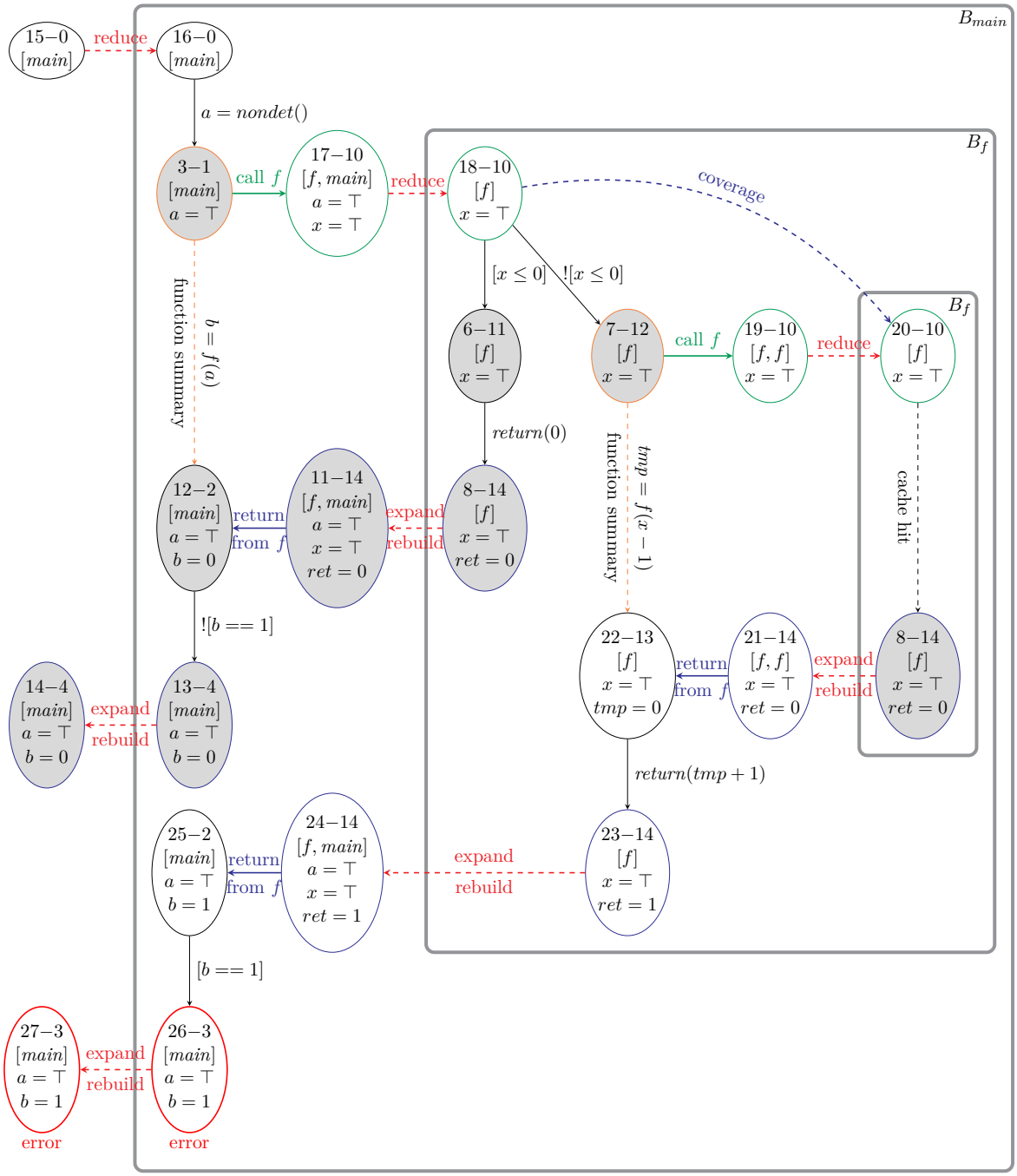


Figure 3.4: ARG produced by the value analysis (second iteration of fixpoint algorithm), each node contains its id and program location in the first line, followed by the call stack and available assignments

stract states in the current analysis, which are not covered by any previously reached abstract states, the fixpoint algorithm has to iterate again. However in this case, the target state is found at the end of the second iteration of the fixpoint algorithm, thus the analysis reports the counterexample and terminates.

3.2 Groundwork and Necessary Preconditions in BAM

This section provides an overview about the preconditions needed for the fixpoint algorithm of BAM to handle recursive procedures.

3.2.1 Most Outer Block

BAM divides the CFA into blocks, which might be nested. The analysis of a block itself is done as part of the transfer relation $\rightsquigarrow_{BAM}$ of BAM. Thus BAM controls the analysis and its result. The previous definition of BAM did not use a most outer block for the whole CFA and the first block started somewhere in the middle of the interprocedural CFA A. This avoids using the operators *reduce* and *expand* of BAM completely if no block is reached during the analysis.

For this thesis a block B_{main} for the interprocedural CFA A was introduced such that B_{main} contains all program locations of the program and thus surrounds all other blocks. The block B_{main} has one input location l_0 with $In(B_{main}) := \{l_0\}$, which is the initial location of the CFA A. If the program terminates, there is one output location l_{exit} with $Out(B_{main}) := \{l_{exit}\}$, which is the exit location of the CFA A, i. e. the program location after the return statement of the main method. If the program does not terminate (perhaps due to an endless loop), there is no output location and $Out(B_{main}) := \emptyset$.

As the block B_{main} includes all nodes of the CFA A, it also includes every possible program location of a target state. If BAM encounters a target state, this abstract state is returned as successor for the current block and transitively also for each wrapping block until the most outer block B_{main}. Thus for the initial abstract state e_{l_0} at the initial program location l_0, the transfer relation $\rightsquigarrow_{BAM}$ of BAM returns either a single target state (from any location inside the program), the abstract states at the exit location l_{exit}, or no abstract state at all (if the program is `SAFE` and not terminating). After the termination of the most outer CPA-Algorithm, the set $reached_{main}$ of that algorithm only consists of the initial abstract state e_{l_0} and its successors for the transfer relation $\rightsquigarrow_{BAM}$ of the block B_{main}.

3.2.2 Blocks for Functions

The definition of a block in a program analyzed with BAM allows to choose blocks and their size very loosely. The original approach of BAM does not distinguish different types of blocks. However as recursion is a property of functions, we now need to handle blocks for functions in a special way.

As described earlier, each function f of a program corresponds to its CFA A_f. A "function block" B_f consists of the body of the function f and includes all further (transitive) function calls in the body. If a function f does not contain calls to further functions, the CFA A_f and the block B_f consist of an identical set of program locations and edges, i. e. for a CFA $A_f = (L_f, l_{f0}, G_f)$ with a function entry location l_{f0} the corresponding block is defined as $B_f = (L_f, G_f)$. If other functions are called from the function f, the blocks for the called functions are part of the current block B_f such that all program locations and edges of nested blocks and corresponding function's call- and return-edges are included in B_f.

The input location of a function block B_f is the function entry l_{f0} of the CFA A_f, i. e. $In(B_f) = \{l_{f0}\}$. The only output location is the function exit l_{exit}, where the control flow merges after executing the function. If there is no function exit (the control flow of the function leads into an endless loop), there is no output location, i. e. $Out(B_f) = \emptyset$. The most outer block B_{main} is the function block of the program's main function (including global declarations).

3.3 Transfer-Relation of BAM with Support for Recursive Procedures

To analyze recursive procedures the transfer relation $\rightsquigarrow_{BAM}$ of BAM, which is defined in Section 2.4.6, is modified to take care of possible recursion when processing function blocks. The new transfer relation $\rightsquigarrow_{BAMrec}$ uses an additional fixpoint algorithm (Algorithm 10), which under-approximates the state space of recursive functions and (implicitly) increments the number of unrollings of recursive function calls until there are no new abstract states during the analysis or a target state is found. This guarantees that either the absence of target states in the program or allows a further analysis of the found error-path.

This section describes the changes in BAM and the fixpoint algorithm. The changes in the algorithms towards the original BAM-CPA $\mathbb{D}_{BAM}$ are highlighted in the algorithms 9, 11 and 12. Then a (partial) proof for soundness of the procedure is given, where the theory of the fixpoint algorithm (Algorithm 10) is analyzed.

Algorithm 9: $\rightsquigarrow_{BAMrec}(e, \pi)$, modified version of Algorithm 3

Input : an abstract state e with a precision π
Output : succeeding abstract states of e
Variables: a program location $l \in L$,
a set $blockResult \subseteq E \times \Pi$

```
1  l := location(e)
2  if blockStack = [] ∧ l = l_0 then
       // use fixpoint algorithm to handle recursive function calls
3      return fixpoint(B_main, e, π, l)
4  else if B_cur ≠ Null ∧ l ∈ Out(B_cur) then
       // leave block B_cur
5      return ∅
6  else if ∃B ∈ ℬ : l ∈ In(B) then
       // enter block B
7      return analyseBlock_rec(B, e, π, l)
8  else
       // forward to wrapped transfer relation
9      return {e' | e ⇝_w (e', π)}
```

Algorithm 10: $fixpoint(B_{main}, e, \pi, l_0)$

Input : the block B_{main},
an abstract state e with a precision π at program location $l_0 \in In(B_{main})$, where l_0 is the initial program location
Output : succeeding abstract states of e
Variables: a set $blockResult \subseteq E \times \Pi$

```
1  while true do
2      recursionFound := false
3      resultStatesStable := true
4      recursionUpdateStates := ∅
5      blockResult := analyseBlock_rec(B_main, e, π, l_0)
6      if {(e_t, π_t) ∈ blockResult | isTargetState(e_t)} ≠ ∅ then
           // target state reached
7          return blockResult
8      if resultStatesStable then
           // fixpoint reached
9          return blockResult
10     foreach e ∈ recursionUpdateStates do
11         addStateToWaitlist(e)
```

The algorithms of BAM use some global variables that are explained here:

- The cache $Cache \subseteq \mathcal{B} \times E \times \Pi \to 2^E \times 2^E$ denotes the central data structure of BAM and is empty before the analysis starts.
- The block $B_{cur} \in \mathcal{B}$ represents the block of the current analysis. Whenever a new sub-analysis for a block starts or terminates, this variable is updated. The initial value of B_{cur} is $Null$.
- The stack *blockStack* is initially empty and consists of triples $\mathcal{B} \times E \times \Pi$, which can be pushed onto the stack or popped from it.
- The boolean variables *recursionFound* and *resultStatesStable* are used as flags in the fixpoint algorithm.
- The set $recursionUpdateStates \subseteq E$ is used to track the (not reduced) initial abstract states from block entries on paths to recursive function calls.

3.3.1 Block Stack and Unrolling Recursive Function Calls

The transfer relation $\rightsquigarrow_{BAMrec}$ maintains a stack *blockStack* that contains the current nesting of blocks in the analysis with their current initial abstract states and precisions. The stack *blockStack* is the basic data structure in the Algorithms 9 and 11 to avoid endless recursion that might be caused by a (too) weak precision or a program with an actual endless recursion. When a block B is entered (and a new analysis is started), a new element (B, e_i, π_i) consisting of the entered block B, the reduced initial state e_i, and its precision π_i is pushed on the stack. After leaving the block B, this element is removed from the stack again.

The fixpoint algorithm aborts the recursive analysis of a block B (started with an initial abstract state e_i and a precision π_i), if the element (B, e_i, π_i) is covered by another element (B', e'_i, π'_i) already located in the stack, i. e. the assumptions $B = B'$ and $e_i \sqsubseteq e'_i$ are valid. If the analysis would not abort the recursive analysis due to the coverage relation, the whole analysis would never terminate, because the abstract state e_i (or another state covered by e'_i) would be found again after the same number of steps in the analysis as there were from e_i to e'_i. The path from e'_i to e_i would repeat and the recursion would be unrolled forever in the current analysis.

When the analysis of a recursive block is aborted because of the coverage relation at a block entry, Algorithm 11 tries to get a previously computed result from the cache to use it as function summary. If the cache does not contain a previous

result, the analysis returns an empty set as result of the function execution. This indicates that the recursion is completely aborted with this function call and only non-recursive paths of the calling function are analyzed further. If a previous result is found in the cache, this result is applied as function summary and returned as result of the function execution. As the function is under-approximated through the function summary from the previous function execution, the analysis (implicitly) increments the number of unwindings of the recursive function, because the overall execution depth of the program is higher in this fixpoint iteration than when analyzing the cached entry in a previous pass.

3.3.2 Fixpoint-Iteration

The previous version (Algorithm 3) of the transfer relation $\rightsquigarrow_{BAMrec}$ performs only three distinct actions depending on the current program location: either enter or leave a block, if the current location is an input or output location of blocks, or just forward to the wrapped CPA $\mathbb{D}_w$ in all other cases. As additional (fourth) case, the transfer relation $\rightsquigarrow_{BAMrec}$ given in Algorithm 9 now also checks for an empty stack at the initial program location l_0. This case only appears at the beginning of the BAM analysis, before any other action is performed. Thus it is guaranteed that the abstract state e_{l_0} is the initial abstract state of the main reached set (for all iterations of the CEGAR loop) and that the most outer block B_{main} is going to be entered.

After entering the block B_{main}, the fixpoint algorithm 10 is executed and iterates until a target state is found or no further abstract states are reachable. In case of a found target state, the fixpoint iteration aborts and the CEGAR-Algorithm will analyze the reached abstract states to get counterexample and perform a refinement if the counterexample is spurious. The second case is the actual fixpoint of the algorithm, because, if no new abstract states are reached in the analysis (and no target state was found), the recursive procedure of the analyzed program is proven to fulfill the specification.

The fixpoint algorithm itself performs BAM analysis internally. As the first operation in the analysis is pushing an element on the *blockStack*, the transfer relation $\rightsquigarrow_{BAMrec}$ will start to analyze the block B_{main}. If any of the loop conditions is satisfied after analyzing the block B_{main}, the fixpoint algorithm returns the result of the current analysis. Otherwise an update of sets $waitlist_B$ for all blocks B is performed, which is similar to the update of the ARGs after a successful precision refinement, where the (not yet reduced) initial abstract states of subtrees are re-added

Algorithm 11: $analyseBlock_{rec}(B, e_I, \pi_I, l)$, modified version of Algorithm 4

Input : a block B that should be analyzed,
an abstract state e_I with precision π_I at a program location $l \in In(B)$
Output : resulting abstract states after analyzing the block B
Variables: the sets *reducedResult*, *expandedResult*, and *blockResult* (each of them $\subseteq E \times \Pi$)

```
1  (e_i, π_i) := reduce(e_I, π_I)
2  blockStack.push((B, e_i, π_i))
3  if isFunctionEntry(l) ∧ l ≠ l_0 then
4  |  if stack contains an element (B, e_c, π_c) with e_i ⊑ e_c then
5  |  |  recursionFound := true
6  |  |  if Cache contains (B, e_c, π_c) then
7  |  |  |  (reached, waitlist) := Cache(B, e_c, π_c)
8  |  |  else
9  |  |  |  (reached, waitlist) := (∅, ∅)
10 |  |  reducedResult := {(e_o, π_o) ∈ reached|
11 |  |                        isTarget(e_o) ∨ location(e_o) ∈ Out(B)}
12 |  else
13 |  |  reducedResult := getReducedResult(B, e_i, π_i)
14 |  expandedResult := {expand(B, e_I, π_I, e_o, π_o)|(e_o, π_o) ∈ reducedResult}
15 |  e_call := getPredecessorStateFromARG(e_I)
16 |  blockResult := {(rebuild(e_call, e_I, e_O), π_O)|(e_O, π_O) ∈ expandedResult}
17 else
   |  // loop-block or B_main
18 |  reducedResult := getReducedResult(B, e_i, π_i)
19 |  blockResult := {expand(B, e_I, π_I, e_o, π_o)|(e_o, π_o) ∈ reducedResult}
20 if recursionFound then
21 |  recursionUpdateStates := recursionUpdateStates ∪ {e_I}
22 blockStack.pop()
23 return blockResult
```

Algorithm 12: $getReducedResult_{rec}(B, e_i, \pi_i)$, modified version of Algorithm 5

Input : a block B that should be analyzed,
a reduced abstract state e_i with a reduced precision π_i
Output : result abstract states of e
Variables: four sets R_0, W_0, $targetStates$, and $blockResult$
(each of them $\subseteq E \times \Pi$)

1 **if** *Cache contains* (B, e_i, π_i) **then**
2 $(R_0, W_0) := Cache(B, e_i, \pi_i)$
3 **else**
4 $R_0 := \{(e_i, \pi_i)\}$
5 $W_0 := \{(e_i, \pi_i)\}$
6 $setBlock(\mathbb{D}_{BAM}, B)$
7 $(reached, waitlist) := CPA(\mathbb{D}_{BAM}, R_0, W_0)$
8 $Cache(B, e_i, \pi_i) := (reached, waitlist)$
9 $targetStates := \{(e_t, \pi_t) \in reached | isTarget(e_t)\}$
10 **if** $targetStates = \emptyset$ **then**
11 $blockResult := \{(e_o, \pi_o) \in reached | location(e_o) \in Out(B)\}$
12 **else**
13 $blockResult := targetStates$
14 **if** *blockResult is not covered by* R_0 **then**
`// fixpoint criteria not reached`
15 $resultStatesStable := false$
16 **return** $blockResult$

to the corresponding sets $waitlist_B$. All abstract states that have been reduced on paths to (recursive) function calls are re-added to their corresponding sets *waitlist*. This procedure is analogue to the lazy refinement procedure that is executed during the refinement in BAM when CEGAR removes an infeasible counterexample. This update guarantees that the whole program is analyzed again and uses entries from the cache as basis for all sub-analysis. This causes the deeper unrolling of a possible recursion. The algorithms re-adds only the minimal set of abstract states that are necessary to re-compute all cache accesses on the current paths. Thus only a few states need to be re-explored in the next iteration of the fixpoint algorithm.

3.3.3 Rebuilding Abstract States at Function-Returns

BAM uses a new operator $rebuild : E \times E \times E \to E$ that augments the expanded function's exit state e_O with information from the function call state e_{call} and the (not reduced) function entry state e_I and returns a new abstract state e_R in order to rebuild knowledge about the outer function scope, which might be invalidated during the function's execution. The operator *rebuild* can, for example, re-assign values from the calling scope to variables that have been be overridden or omitted in the called function due to equal names and thus have an invalid value. For a sound analysis the operator must resolve all collisions of equal-named variables in a sound way. In Algorithm 11 the operator *rebuild* is applied in line 16. For convenience we define an operator $rebuild_{id}(e_{call}, e_I, e_O) := e_O$ that returns the identity of the abstract output state and can be used by analyses that do not need to re-add information at function exits.

Figure 3.5 gives an overview of the application of the operators *reduce*, *expand*, and *rebuild* in BAM. The abstract state e_{call} is the abstract state right before entering the function, i. e. before assigning all parameters of the function (shown as $p := a$). There is only one abstract state e_{call}, from which e_I is reachable. In Algorithm 11 the function $getPredecessorStateFromARG$ is used to extract e_{call} from the ARG, which tracks all relations between abstract states. In Figure 3.5 the abstract state e_I is located at the function-entry-node of a procedure call $b := f(a)$ with an argument a and a target variable b for the return value. The function block B_f itself is highlighted. The abstract states e_R with precision π_O are the abstract successor states of e_I regarding the transfer relation $\rightsquigarrow_{BAM}$ of BAM. This is necessary for predicate analysis to handle recursive procedures.

Figure 3.6 shows the relation between the operators *reduce*, *expand* and *rebuild*. As extension to Figure 3.5 it also contains the precisions for all abstract states. In

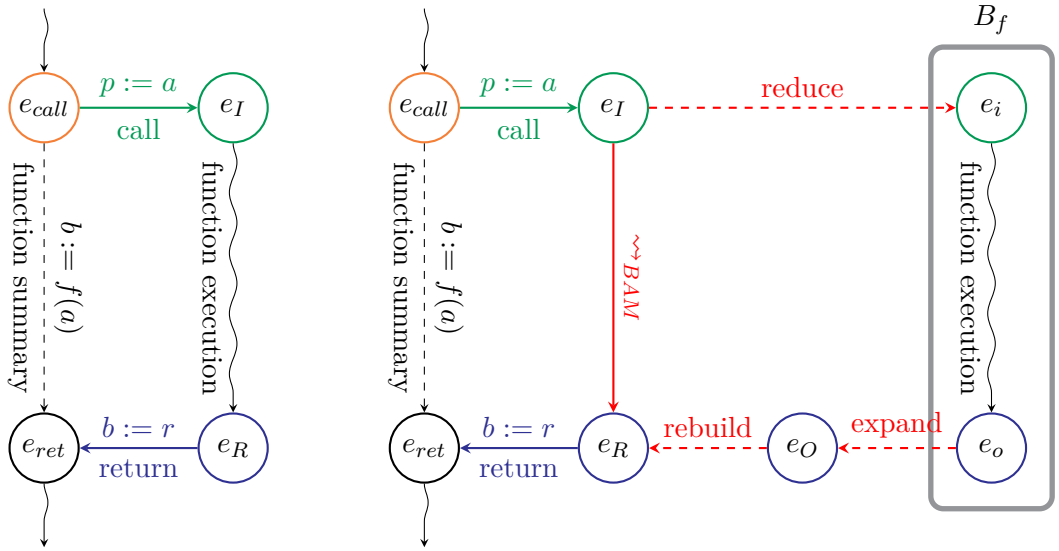


Figure 3.5: Default function execution versus execution of function block with operators *reduce*, *expand* and *rebuild*

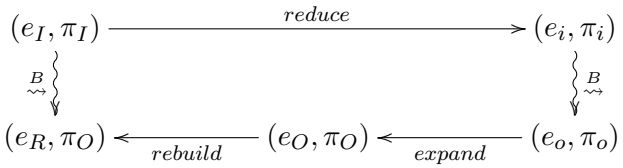


Figure 3.6: Schematic diagram of the operators *reduce*, *expand* and *rebuild*

favor of a simple implementation of the transfer relation, we rebuild the abstract states after every function call and not only after recursive procedures. The rebuilding could be restricted to recursive function calls only as possible optimization.

3.4 Theory and Outline for the Proof of Correctness

This section describes the theory behind the fixpoint algorithm and provides an outline of its correctness. The proof is based on Hoare's rules, which are described in Section 2.5.2. The fixpoint algorithm itself only computes the point, when further unrolling of the recursive function does not lead to new abstract states at the recursive function's exit, because all abstract states of the recursive function's entry and exit are covered.

For simplicity the precision of abstract states is ignored in this section. As the

precision is only changed during the refinement step of CEGAR, the fixpoint algorithm only uses a precision that is constant for each program location in the analysis. The operators *reduce* and *expand* are omitted in this section, because the important steps of the proof are based on complete analyses of blocks. We assume that the call stack is implicitly reduced and expanded according to the function calls. The proof only uses the abstract states after the application of *reduce* at the function's entry and the abstract states after expanding the block's output states at the function's exit. Additionally the operator *rebuild* is not applied in the proof. It only removes invalid information from abstract states at the function exit location and as this proof uses the result of this step, i. e. valid abstract states, the operator is applied implicitly as last step of a block's analysis and can be omitted for the coverage relations of this proof. Thus also conflicts of identifiers are not important in this proof, because renaming of variables or other steps that avoid collisions of identifiers are performed with the application of the operator *rebuild*.

3.4.1 Hoare's Rules and Abstract States

The concretizations $[[P_e]]$ and $[[Q_e]]$ of sets P_e and Q_e of abstract states match the pre- and postcondition P and Q of a statement T. A Hoare-Triple $\{[[P_e]]\}T\{[[Q_e]]\}$, consisting of a program statement T and two concretizations $[[P_e]]$ and $[[Q_e]]$, indicates that, if T is started with a concretization of a set P_e of abstract states and T terminates, then the result is a subset of the concrete states of Q_e after T completes.

Under - and Over-Approximation of Abstract States

A set S' of abstract states under-approximates another set S of abstract states, if all concrete states $[[S']]$ are a subset of the concretization $[[S]]$, formally $[[S']] \subseteq [[S]]$. The case $[[S']] \supseteq [[S]]$ is called over-approximation of S.

The rule of consequence allows to under-approximate (strengthen) the precondition $[[P_e]]$ and to over-approximate (weaken) the postcondition $[[Q_e]]$ of a statement T. With this notation the rule of consequence can also be written as:

$$\frac{[[P'_e]] \subseteq [[P_e]] \qquad \{[[P_e]]\}T\{[[Q_e]]\} \qquad [[Q_e]] \subseteq [[Q'_e]]}{\{[[P'_e]]\}T\{[[Q'_e]]\}} \text{ CONSEQUENCE}$$

Function Instantiation

Hoare's rule for function instantiation also can be applied to abstract states. As assignments of parameters and return values are done by the corresponding analysis,

they are omitted in the rule. A function f is called with abstract states P_e at the function entry, thus also the function body is directly entered with the abstract state P_e. The function body B_f exits with abstract states Q_e, so the resulting abstract states after the function execution are Q_e.

$$\frac{\{[[P_e]]\}B_f\{[[Q_e]]\}}{\{[[P_e]]\}b = F(a)\{[[Q_e]]\}} \text{ FUNCTIONCALL}$$

The abstract states P_e and Q_e correspond to the function entry and function exit locations of a CFA, i. e. the assignment $p := a$ of parameters happens before reaching P_e and the assignment of return values $b := r$ appears after Q_e. Thus all assignments for parameters and return values are not part of this step, but are handled by the underlying analyses in a sound way.

Recursive Procedures

Hoare's rule of recursion indicates that, if the body B_f of a function f satisfies (the concretizations of) abstract input states P_e and abstract output states Q_e under the condition, that all recursive calls to the function f also satisfy P_e and Q_e, then P_e and Q_e also are valid abstract states for the whole function. The assignments of parameters and return values for the function call are done implicitly through another analyses and are omitted here.

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

3.4.2 Soundness of the Fixpoint Algorithm

A program analysis is sound if it never misses a property violation. The fixpoint algorithm terminates either in case of a property violation or when there was no new abstract state reachable in the last iteration. To prove the soundness of the fixpoint algorithm, we only concentrate on the second case, because in the first case, when the property violation is found, the fixpoint algorithm aborts and returns the target state.

As recursive procedures should only be unrolled as little as needed, but as often as necessary, Hoare's rule for recursion is used as basis to know, when the fixpoint algorithm reaches this limit and further unrolling can be aborted. In the following the index k denotes the current iteration of the fixpoint algorithm.

For the theory we assume a function f with an initial entry state $e_i \in E$. All paths through f are analyzed, except paths containing a recursive call to f with an

entry state e_i' with $e_i' \sqsubseteq e_i$. If a function call to a function g with a not covered entry state e_i'' is reached, the function g is analyzed as far as possible. The result at the output location $Out(B_f)$ of the function block B_f is a set $E_k \subseteq E$ of abstract states. If there is not a single non-recursive path through the function f, it contains an endless recursion and the set E_k is empty, because the output location $Out(B_f)$ is never reached. However this does not influence the further proof.

The fixpoint loop is executed at least once, such that at least one non recursive path is analyzed completely. The next fixpoint iteration re-analyses the block B_f with the initial entry state e_i. If the recursive call with a covered entry state $e_i' \sqsubseteq e_i$ is reached again, the function execution of the previous fixpoint iteration is used to summarize the current function call. The re-usage is possible due to the cache of BAM that contains the abstraction of the previously analyzed function block. The over-approximation $e_i \sqsupseteq e_i'$ guarantees the soundness of using e_i instead of e_i' as initial abstract state in the block B_f and the partial correctness of using the block's abstract states as function instantiation is also given by the rules of Hoare:

$$\dfrac{\dfrac{\text{over-approx. } [[e_i']] \subseteq [[e_i]] \qquad \text{abstraction from cache } \{[[e_i]]\}B_f\{[[E_k]]\} \qquad \text{identity } [[E_k]] \subseteq [[E_k]]}{\{[[e_i']]\}B_f\{[[E_k]]\}}\ \text{CONSEQUENCE}}{\{[[e_i']]\}b = f(a)\{[[E_k]]\}}\ \text{FUNCTIONCALL}$$

function summary

The block B_f is analyzed and left with a set $E_{k+1} \subseteq E$ of abstract states. The fixpoint algorithm computes the abstract states $E_{new} \subseteq E_{k+1}$ that are not covered by any previous abstract states $e_k \in E_k$, formally $E_{new} := \{e_{k+1} \in E_{k+1} | \nexists e_k \in E_k : e_i \sqsubseteq e_k\}$. The fixpoint is reached if E_{new} is empty (for each analyzed function of the program), i. e. if all abstract states of E_{k+1} are covered by abstract states of E_k. This coverage relation can be expressed with concretizations as $[[E_{k+1}]] \subseteq [[E_k]]$. In this case the following rule of consequence is satisfied:

$$\dfrac{\text{identity } [[e_i]] \subseteq [[e_i]] \qquad \text{analysis of the block } \{[[e_i]]\}B_f\{[[E_{k+1}]]\} \qquad \text{coverage } [[E_{k+1}]] \subseteq [[E_k]]}{\{[[e_i]]\}B_f\{[[E_k]]\}}\ \text{CONSEQUENCE}$$

transfer relation

When the fixpoint is reached and both rules given above are satisfied, i. e. the fixpoint algorithm used a former abstraction to summarize the recursive function call and executed the analysis for the current function block resulting in no new abstract states, the rule of recursion can be applied and prove the (partial) correctness of the fixpoint algorithm:

$$\frac{\overset{\text{function summary}}{\{[[e_i]]\}b = f(a)\{[[E_k]]\}} \quad \vdash \quad \overset{\text{transfer relation}}{\{[[e_i]]\}B_f\{[[E_k]]\}}}{\{[[e_i]]\}b = f(a)\{[[E_k]]\}}\ \text{RECURSION}$$

When the fixpoint algorithm terminates, these rules are fulfilled. Thus the soundness of the approach is shown. □

4 Using Further Analyses in Combination with BAM

This chapter describes the some analyses that can be combined with BAM. As first some basic analyses are considered that are necessary to determine or track the current program location, the call stack and dependencies between abstract states like coverage, merges and transfers. Then two more precise analyses - the value analysis and the predicate analysis - are used to show the working concept of BAM. Both analyses are used as component in a CEGAR-loop and thus also the refinement strategies and their precisions will be examined.

The previously explained fixpoint algorithm (Algorithm 10) in BAM allows to abstract from an underlying program analysis (the wrapped CPA $\mathbb{D}_w$) for recursive procedures. Such a program analysis has to do the heavy work, for example the tracking of variables and values. As most analyses identify variables with their name (including their current scope, i. e. the function's name), they have to handle collisions of the identifiers that appear with recursive procedures, without omitting the soundness of the analysis and the chance of re-using a block in BAM. The identifier for a local variable x in a procedure f might be "f::x". In this case there is no difference for the analysis between the variable in different function calls, even if one identifier hides the other one in a recursive call of f. An analysis might use indices of a static single assignment (SSA) to distinguish variables with equal identifiers, but this avoids the re-use of verification results between the blocks of BAM and causes problems with the operations *merge* and *stop* as they have to work on abstract states containing information with different indices.

In the following, the solution for the problem with the identifiers is explained for the value analysis and the predicate analysis. We also show that the implemented operator *rebuild* in the described analyses is sound.

4.1 ARG-CPA

The ARG-CPA $\mathbb{D}_{ARG} = (D_{ARG}, \rightsquigarrow_{ARG}, merge_{ARG}, stop_{ARG})$ tracks relations between abstract states, such as transfer relation, merge and coverage. Similar to the BAM-CPA it also is implemented as a wrapper around another CPA $\mathbb{D}_w$ that does the main part of the analysis. The operators $\rightsquigarrow_{ARG}$, $merge_{ARG}$, $stop_{ARG}$, and also the BAM-specific operators $reduce_{ARG}$, $expand_{ARG}$, and $rebuild_{ARG}$ just forward to their counterparts in the CPA $\mathbb{D}_w$. As mentioned before, the ARG-CPA is a main component of BAM, because the ARG tracks dependencies between abstract states, and BAM needs to know about them to compute error paths for refinements.

4.2 Location-CPA

The Location-CPA $\mathbb{D}_{loc} = (D_{loc}, \rightsquigarrow_{loc}, merge_{loc}, stop_{loc})$ tracks the current program location. An abstract location state e_{loc} contains exactly one program location $l \in L$, i. e. a node of the interprocedural CFA. The initial location state for the CPA-Algorithm represents the initial program location l_0.

The abstract domain $D_{loc} = (C, \mathcal{E}_{loc}, [[\cdot]])$ consists of the set C of concrete states, the lattice $\mathcal{E}_{loc} = (E_{loc}, \sqsubseteq, \sqcup, \top, \bot)$ and a concretization function $[[\cdot]]$. The lattice $\mathcal{E}_{loc}$ uses the set $\mathcal{L} = L \cup \{\top_{\mathcal{L}}, \bot_{\mathcal{L}}\}$ that induces the (flat) lattice over program locations with the least upper bound $\top_{\mathcal{L}}$ and the greatest lower bound $\bot_{\mathcal{L}}$. The transfer relation $\rightsquigarrow_{loc}$ returns the set $\{l' | (l, op, l') \in G\}$, which contains all nodes reachable from l via a CFA edge. The default operators $merge_{sep}$ and $stop_{sep}$ serve as $merge_{loc}$ and $stop_{loc}$. The operators $reduce_{loc}$ and $expand_{loc}$ return the identity of the given abstract states, because the program location must always be tracked. Thus we set $reduce_{loc} := reduce_{id}$ and $expand_{loc} := expand_{id}$. The operator $rebuild_{loc}$ re-adds lost information about the outer function scope. Because the LocationCPA does not lose information through a function call, the operator $rebuild_{loc} := rebuild_{id}$ is defined to also return the identity of the given abstract state.

4.3 Callstack-CPA

The Callstack-CPA $\mathbb{D}_{call} = (D_{call}, \rightsquigarrow_{call}, merge_{call}, stop_{call})$ tracks the current call stack. An abstract call stack state e_{call} consists of the current scope information in form of a call stack, i. e. a stack $[f_1, f_2, ...f_n]$ of all currently available function scopes, where f_n is the scope of the current program location and was called from f_{n-1}. If the current edge in the analysis is a function call, the called function is appended the

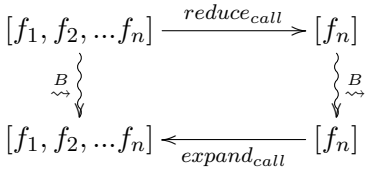


Figure 4.1: Schematic diagram of the operators $reduce_{call}$ and $expand_{call}$

call stack of the abstract state. In case of a function return, the left function is removed from the call stack of the abstract state. Similar to the Location-CPA the operators $merge_{call}$ and $stop_{call}$ are $merge_{sep}$ and $stop_{sep}$, respectively.

The operators $reduce_{call}$ and $expand_{call}$ can return the identity of the given input for the default BAM analysis, however it necessary for handling recursive procedures that they actually change the call stack. Therefore the operators are defined as

$$reduce_{call}(B, [f_1, ...f_n], \pi_I) := ([f_n], \pi_I) \text{ and}$$

$$expand_{call}([f_1, ...f_{n-1}], \pi_I, B, [f_n], \pi_o) := ([f_1, ...f_n], \pi_o).$$

The operator $reduce_{call}$ removes the tail $[f_1, ...f_{n-1}]$ of the abstract call stack state $e_I = [f_1, ...f_n]$, only the most local function scope f_n remains on the stack. The operator $expand_{call}$ concatenates the previously removed tail $[f_1, ...f_{n-1}]$ of the abstract input state e_I with the call stack $[f_n]$ of the abstract output state $e_o = [f_n]$ to get a complete stack again. Figure 4.1 gives an overview of the operators.

The operators $reduce_{call}$ and $expand_{call}$ are independent of the current type of block: both function and loop blocks are handled equally. If the current program location is part of a nested block, the expanded call stack (after $expand_{call}$ was applied) contains only levels up to the next outer block. With function scopes as block-size, the stack has never more than two level, because after each function call, the operator $reduced_{call}$ is applied to the call stack to get only the level of the called function and before a function exit, the operator $expand_{call}$ re-adds only the level of the calling scope as tail of the current call stack. The Callstack-CPA only allows blocks, where input and output location belong to the same function, because there is no support for blocks with different input and output call stacks.

The operator $rebuild_{call} := rebuild_{id}$ returns the identity of the given abstract state. This might be surprising, as recursion makes heavy usage of the call stack in a program. However it is possible to distinguish different levels on the call stack, even if the called functions have equal names. Thus no information is corrupted through recursive procedures and has to be rebuild.

4.4 Value Analysis

The value analysis is a more precise analysis and tracks variables with their explicit numerical values. A detailed formal definition of the value analysis was written by Beyer and Löwe [9], so beside the BAM-specific parts only the basic components are described here.

4.4.1 Value-CPA

The Value-CPA $\mathbb{D}_{val} = (D_{val}, \Pi_{val}, \rightsquigarrow_{val}, merge_{val}, stop_{val}, prec_{val})$ implements the value analysis and provides a refinement procedure for CEGAR.

Abstract Domain The abstract domain $D_{val} = (C, \mathcal{E}_{val}, [[\cdot]])$ consists of the set C of concrete states, the lattice $\mathcal{E}_{val} = (E_{val}, \sqsubseteq, \sqcup, \top, \bot)$, and a concretization function $[[\cdot]]$. The lattice $\mathcal{E}_{val}$ uses the set $E_{val} = (X \to \mathcal{Z}) \subset X \times \mathcal{Z}$ of abstract variable assignments with a set X of variables and a set $\mathcal{Z} = \mathbb{Z} \cup \{\top_{\mathcal{Z}}, \bot_{\mathcal{Z}}\}$ of values, which induces the (flat) lattice over numbers with the least upper bound $\top_{\mathcal{Z}}$ and the greatest lower bound $\bot_{\mathcal{Z}}$. This definition here uses only the set $\mathbb{Z}$ of integers, but other numeral types like floats or rationals are also possible in theory and supported in implementation. Let $e(x)$ denote the value of a variable $x \in X$ in the assignments of $e \in E_{val}$, i. e. $e(x) = y$ is equal to $(x, y) \in e$. The partial order $\sqsubseteq \subseteq E_{val} \times E_{val}$ is given as $e \sqsubseteq e'$, if the following expression is satisfied: $(\forall x \in X : e(x) = e'(x) \vee e(x) = \bot_{\mathcal{Z}} \vee e'(x) = \top_{\mathcal{Z}})$.

Merge and Stop Operator The operators $merge_{val}$ and $stop_{val}$ are $merge_{sep}$ and $stop_{sep}$, respectively.

Precision A precision $\pi \in \Pi_{val} = 2^{L \times X}$ specifies a set of variables and corresponding program locations, where the variables should be tracked, i. e. some variables are only tracked certain program locations.

Transfer Relation The transfer relation $\rightsquigarrow_{val}$ evaluates all expressions of an analyzed edge of the CFA and updates the abstract state accordingly. The evaluation $eval(exp, e)$ of an expression exp with an abstract value state $e \in E_{val}$ is defined as

$$eval(exp, e) = \begin{cases} \bot_{\mathbb{Z}}, & \text{if } \exists y \in X : (y, \bot_{\mathbb{Z}}) \in e \\ \top_{\mathbb{Z}}, & \text{if } \exists y \in X : (y, \top_{\mathbb{Z}}) \in e \wedge (y \text{ occurs in } exp) \\ v, & \text{otherwise, where } exp \text{ evaluates to } v \text{ after substituting} \\ & \text{all variables } x \in X \text{ in } exp \text{ with } e(x) \end{cases}$$

An assignment of an evaluation $eval(exp, e)$ to a variable $x \in X$ at a program location $l \in L$ with a precision π is defined as

$$e(x) = \begin{cases} eval(exp, e), & \text{if } (l, x) \in \pi \\ \top_{\mathbb{Z}}, & \text{otherwise.} \end{cases}$$

The assignment overrides the value of the variable x in the abstract state e if x should be tracked at the program location l.

4.4.2 Reduce and Expand

The operators $reduce_{val}$ and $expand_{val}$ for the value analysis are defined based on the scope of the corresponding block. If a variable is never accessed (read- or write-access) in a block B, the $reduce_{val}$ operator removes the assignment of the variable from the initial abstract state e_I and $expand_{val}$ re-adds it to the output states e_o of the block. For example all variables except the parameters of the called function and global variables can be removed with the reduction in case of a function block, because they are out of scope. The precision is reduced and expanded similarly according to the usage of variables in the block.

The formal definition of the operators $reduce_{val}$ and $expand_{val}$ is:

$$reduce_{val}(B, e_I, \pi_I) = (\{(x, e_i(x)) \in e_I | x \text{ used in } B\}, \{(l, x) \in \pi_I | x \text{ used in } B\})$$

$$expand_{val}(e_I, \pi_I, B, e_o, \pi_o)) = \begin{pmatrix} \{(x, e_I(x)) \in e_I | x \text{ not used in } B\} \cup e_o, \\ \{(l, x) \in \pi_I | x \text{ not used in } B\} \cup \pi_o \end{pmatrix}$$

Proof of Soundness

To proof soundness of the operators $reduce_{val}$ and $expand_{val}$ we have to show that the expanded output states over-approximate a direct analysis of the block without reducing and expanding states. For the value analysis it can even be proven that the abstract states are not only over-approximating, but identical for analyses with BAM and without, if the same abstract state and precision is given as input for both cases.

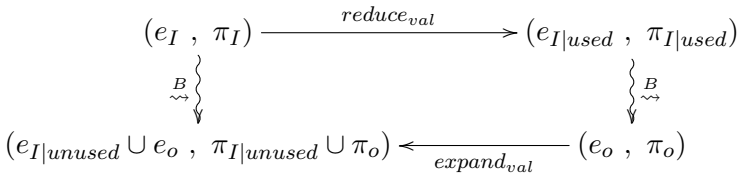


Figure 4.2: Schematic diagram of the operators $reduce_{val}$ and $expand_{val}$

The variable assignments e_I of an abstract value state can be divided into two disjoint partitions $e_{|used}$ and $e_{|unused}$ (i. e. $e_I = e_{I|used} \cup e_{I|unused}$ and $\emptyset = e_{I|used} \cap e_{I|unused}$) according to the access to variables in the block. The partition $e_{I|used}$ contains the assignments of variables that are read or written in the block, the other one consists of all unused variables along the block. Both partitions can be analyzed separately, because there is no data-flow between them. If we ignore pointers and arrays, there is no common memory access for the partitions. The precision π_I can also be split up into two disjoint partitions $\pi_{I|used}$ and $\pi_{I|unused}$ according to the access to variables in the block.

In Figure 4.2 the operators $reduce_{val}$ and $expand_{val}$ and their effects are shown in a simplified manner. The partition $e_{I|used}$ is changed into $e_{o|used}$ during the analysis of the block. With and without BAM the value analysis performs the same operations, computes the same new assignments, and overrides existing ones, because the assignments and the precision are equal for all variables that are accessed for reading or writing. The other partition $e_{I|unused}$, which contains the unused variables, remains unchanged without BAM, because not a single read- or write-access is performed. With BAM the partition $e_{I|unused}$ is removed with $reduce_{val}$, i. e. all assignments for those variables are first set to $\top_{\mathcal{Z}}$ with $reduce_{val}$. After analyzing the block the operator $expand_{val}$ re-adds $e_{I|unused}$ to the abstract state. This sets those assignments to their original value. As the block's analysis does only access variables of the block, the reduced precision $\pi_{I|used}$ is sufficient for the analysis. Thus both analyses return equal abstract states after leaving the block and the operators $reduce_{val}$ and $expand_{val}$ are sound.

4.4.3 Rebuild

The operator $rebuild_{val}$ handles collisions of identifiers as it might happen for each (local) variable of a recursive function. As the value analysis simply tracks variables in a map, the rebuilding is only an update of the map, where the correct assignments are restored. The rebuild abstract state is based on the expanded function return

state e_O and the calling state e_{call}. The values of global variables and the value for the function return variable (special variable for tracking the return value of a function call) are taken unchanged from the expanded function return state e_O, because these variables survive the function scope and their identifiers are surely not colliding, as global variables have an unique identifier in all function scopes and the return variable only exists for the short life time of the function exit. All other values are copied from the calling state e_{call}. The precision π_O remains unchanged.

The formal definition of the operator $rebuild_{val}$ reads as follows:

$$\begin{aligned} rebuild_{val}(e_{call}, e_I, e_O) =& \{(x, e_{call}(x)) \in e_{call} | \neg(isGlobal(x) \vee isRet(x))\} \\ & \cup \{(x, e_O(x)) \in e_O | isGlobal(x) \vee isRet(x)\} \end{aligned}$$

The functions $isGlobal(x)$ and $isRet(x)$ check, if an identifier x represents a global variable or the return variable of the called function, respectively.

4.4.4 Counterexample and Refinement

The value analysis uses a single path (a sequence of abstract states) as counterexample that is checked for feasibility with a concrete execution of the path (i. e. all variables are tracked). In case of an infeasible (spurious) counterexample during the CEGAR-loop, the value analysis uses an interpolation based refinement strategy [9] to adjust its precision. The full description and theory of the interpolation strategy will be omitted here in favor of a short description of only the important part for (recursive) function calls.

As the value analysis guarantees that the counterexample is always a single path in the program, the refinement strategy only has to handle a sequence of program statements. The feasibility check and the refinement strategy analyze the counterexample several times with distinct precisions. Each precision allows to track a subset of variables and the target of the refinement is to find a (minimal) set of variables for each program location needed to exclude the previously found target state from re-exploration. To handle recursive procedures during the path's analysis, the call stack has to be tracked explicitly along the counterexample path and the operator $rebuild_{val}$ must be executed at each return location of a (recursive) function in order to restore the assignments of variables in the calling function scope. As the precision depends on the program location, a variable is either not tracked, because it is not needed in the refinement to proof the counterexample as spurious, or not only tracked in the current function scope, but also in all following (recursive)

function calls along the counterexample, as the same program locations appear in the functions. Thus we might track a variable in more than the necessary program locations, however this is just computational overhead and does not lead to unsound results. This approach always yields a valid refined precision such that the infeasible counterexample is not re-explored in a further CEGAR-iteration.

4.5 Predicate Analysis

The predicate analysis [19, 25] uses predicates and formulae to track variables and values. The refinement procedure is based on Craig interpolation of boolean formulae and the abstraction computation is done with boolean predicate abstraction. In this context there are two different definitions of "block": for the predicate analysis "abstraction blocks" are the set of program locations between two abstraction locations, and in BAM a "block" represents the set of program location bounded by the operators *reduce* and *expand*. We will never use abstraction blocks, but only the blocks of BAM in the following parts.

The predicate analysis uses Adjustable Block Encoding (ABE) [4, 8] to decide where to compute abstractions. To work with BAM the predicate analysis is configured to use at least all input and output locations of all blocks as abstraction locations. Additionally the predicate analysis may also use further program locations as abstraction points, like loop heads or target locations.

As this analysis is more complex, several changes were made during this thesis, which aim to have a sound predicate analysis for recursive programs. In the following first a short description of basic components of the Predicate-CPA and the refinement process is given, then the BAM-specific parts like the operators $reduce_{pred}$, $expand_{pred}$ and $rebuild_{pred}$ follow.

4.5.1 Predicate-CPA

The predicate analysis was developed in the context of CEGAR and is implemented as Predicate-CPA $\mathbb{D}_{pred} = (D_{pred}, \Pi_{pred}, \rightsquigarrow_{pred}, merge_{pred}, stop_{pred}, prec_{pred})$. The Predicate-CPA does not track program locations, but relies on being combined with the Location-CPA such that the program location and the edges of the CFA are available if necessary.

Abstract Domain The abstract domain $D_{pred} = (C, \mathcal{E}_{pred}, [[\cdot]])$ consists of the set C of concrete state, the lattice $\mathcal{E}_{pred} = (E_{pred}, \sqsubseteq, \sqcup, \top, \bot)$, and a concretiza-

tion function $[[\cdot]]$. The set of quantifier-free predicates of program variables is denotes by $\mathcal{P}$. The lattice elements $(\psi, \varphi) \in E_{pred}$ consist of an abstraction formula ψ, which is a boolean combination of predicates from $\mathcal{P}$, and a path formula φ that represents the path (or the disjunction of several paths) from the last abstraction location to the current program location. The top element of the lattice $\mathcal{E}$ is $\top = (true, true)$. If the current program location is an abstraction location, the abstract state is called an abstraction state with a path formula $\psi = true$. The partial order $\sqsubseteq \subset E \times E$ for two elements $e = (\psi, \varphi)$ and $e' = (\psi', \varphi')$ is given as $e \sqsubseteq e' \iff (\psi \Rightarrow \psi' \wedge \varphi \Rightarrow \varphi')$. The operator $\sqcup : E \times E \to E$ returns the least upper bound according to $e \sqcup e' = (\psi, \varphi) \sqcup (\psi', \varphi') = (\psi \vee \psi', \varphi \vee \varphi')$. The operator $join_{pred}$ is based on the operator $\sqcup$ and yields

$$join_{pred}(e, e') = \begin{cases} e \sqcup e', & \text{if } isSameAbs(e, e') \\ e, & \text{otherwise,} \end{cases}$$

where $isSameAbs(e, e')$ checks if the abstractions formulae ψ and ψ' of the abstract states $e = (\psi, \varphi)$ and $e' = (\psi', \varphi')$ are equal and also computed at the same program location. As paths starting in different abstraction states are never merged with this operator, the abstraction states form a tree with the initial abstract state as root, and a path from the initial abstract state to a possible target state consists of a linear chain of abstraction states, where each of them is only reachable from its direct predecessor.

Precision A precision $\pi \in \Pi_{pred} = 2^{L \times \mathcal{P}}$ is a location-mapped set of predicates over the variables of the program, i. e. at each program location different predicates can be used. For a precision π the set of predicates for a certain program location l is denoted with $\pi(l) := \{p | (l, p) \in \pi\}$. The precision is only important for abstraction locations and does not contain predicates for other program locations. The predicates of the precision are determined through interpolation of boolean formulae during the refinement, where each interpolant is split up into its predicates, which are defined as the smallest boolean parts (atoms) of a formula. For example the three predicates $a = 3$, $b = 4$, and $c > d$ would be retrieved from an interpolant $a = 3 \wedge (b = 4 \vee c > d)$.

Transferrelation The transfer relation $\rightsquigarrow_{pred}$ determines the abstraction formula ψ' and the path formula φ' of the next abstract state (ψ', φ') based on the last abstract state (ψ, φ). The edges of a CFA (for example assignments or assumptions) are encoded as boolean formulae with SSA-indices, i. e. a new index for each assigned

variable. The conjunction of such formulae is denoted as path formula. A path formula might refer to the same variable several times, but maybe with distinct SSA-indices to distinguish their relations at different edges of the CFA. Abstraction formulae do not have SSA-indices and need to be "instantiated", when used in computations with path formulae, i. e. in this case SSA-indices are added according to the current context. For calculations including only other abstraction formulae (for example to check coverage) no SSA-indices are used. For simplicity we assume that the index-management is done implicitly and do not provide further distinction here.

For a boolean formula α the strongest post operator $SP_{op}(\alpha)$ defines the strongest boolean formula that is satisfied after the execution of the operation *op*. Thus the strongest post operator for a boolean formula α and a path formula φ is given as $SP_{\varphi}(\alpha) = \alpha \wedge \varphi$ and represents executing successive edges of the CFA starting with an initial condition α. A boolean predicate abstraction for a formula β is denoted with $\beta^{\pi(l)}$ at an abstraction location $l \in L$. The transfer relation $e \overset{op}{\rightsquigarrow} e'$ defines the successor state e' of an abstract state $e = (\psi, \varphi)$ after the operation *op* as

$$e' = \begin{cases} ((\psi \wedge SP_{op}(\varphi))^{\pi(l)}, true), & \text{if } isAbs(e') \\ (\psi, SP_{op}(\varphi)), & \text{otherwise,} \end{cases}$$

where l is the program location of the succeeding abstract state and $isAbs(e')$ checks whether e' is an abstraction state. Depending on the program location l either a new abstraction formula is computed and the path formula is reset to *true* or only the path formula is updated with the strongest post operator.

4.5.2 Predicate Abstraction and Refinement with Interpolation

Predicate abstraction computes summaries for path formulae based on the current precision, which is a set of predicates extracted from interpolants. Therefore the predicate analysis computes a boolean predicate abstraction at each abstraction location by searching for the strongest boolean combination of predicates from $\pi(l)$ that is valid at the abstraction location l. The dependency between the abstraction computation and the interpolation procedure is considered here.

Usage of Inductive Sequences of Interpolants in Predicate Analysis

In case of a found target state, the error path from the initial program location l_0 to the target location l_e is converted into a sequence φ_1, φ_2,... φ_n of path formulae

that reach from one abstraction location to the next one along the path. If the conjunction $\bigwedge_{i=1}^{n} \varphi_i$ of the path formulae is unsatisfiable, the counterexample is spurious. In this case an inductive sequence ψ_1, ψ_2,... ψ_{n-1} of interpolants is computed for the path formulae. The interpolants correspond to the abstract states at abstraction locations and contain all information to exclude the path from re-exploring in the next iteration of CEGAR. The predicate analysis splits the interpolants ψ_i into predicates, which are added to the precisions π_i at the corresponding abstraction locations along the error path. In the next iteration of CEGAR the transfer relation $\rightsquigarrow_{pred}$ uses the predicates from the interpolants for the abstraction formulae.

Proof of Soundness and Progress for the Predicate Analysis

This proof only covers the standard predicate analysis and also its BAM-specific operators *reduce* and *expand*, because this part is based sequential interpolation. The extension with the operator *rebuild* to handle recursive procedures is considered later, as it depends on a different interpolation strategy.

Given two succeeding abstraction formulae ψ_{Abs} and ψ'_{Abs} and a path formula φ that represents the path from the first abstraction location to the second one, the implication $SP_\varphi(\psi_{Abs}) \Rightarrow \psi'_{Abs}$ must be satisfied to guarantee soundness of the predicate analysis, i. e. inductive invariants are needed. An abstraction formula ψ'_{Abs} does never contain more information than provided by the previous abstraction formula ψ_{Abs} and a path formula φ. Thus it will not exclude any possible feasible counterexample from exploration. The abstraction procedure is defined in a way that the boolean predicate abstraction satisfies this criteria, because only important and valid facts are used to build the abstraction formula ψ'_{Abs}. Predicates that are invalid (for example $x = x + 1$) or consist of identifiers that are out of scope at the current program location are filtered out during the abstraction procedure. In most cases the interpolation procedure avoids interpolants that cause such useless predicates. However, invalid predicates might appear in case of recursive procedures, if sequential interpolation is used for the refinement. Thus a different interpolation strategy (like tree interpolation) is used there.

To ensure progress of CEGAR, i. e. no repeated counterexample is found and causes endless iteration in the CEGAR-loop, the abstraction formulae must be strong enough to exclude a found counterexample from re-exploration. Each abstraction formula ψ_{Abs} that is produced with boolean predicate abstraction from predicates $\pi(l)$ is stronger than (or equal to) the corresponding interpolants ψ_{Itp}, where the predicates $\pi(l)$ were extracted from, because an abstraction formula is defined as

the strongest boolean combination of predicates. If the implication $\psi_{Abs} \Rightarrow \psi_{Itp}$ is not fulfilled, the CEGAR-iteration finds the same counterexample again and again. An inductive sequence of interpolants has the property that each interpolant of the sequence is sufficient for the infeasibility of the rest of the path. Thus the corresponding abstraction formula contains enough information to avoid re-exploring the same counterexample.

4.5.3 Reduce and Expand

The predicate analysis is configured to compute boolean predicate abstractions at the input and output locations $In(B)$ and $Out(B)$ of each block B. This causes corresponding path formulae to be *true* and allows to use the abstraction formula of the reduced initial abstract state e_i as identifier for the block's analysis in the cache. The operators $reduce_{pred}$ and $expand_{pred}$ use a heuristic strategy that analyses the block B and the initial precision π_I to divide the set π_I of available predicates into two disjoint partitions defined as $p_{I|used} = \{(l,p) \in \pi_I | l \in B \wedge p \text{ used in } B\}$ and $p_{I|unused} = \pi_I \setminus \pi_{I|used}$. The partition $\pi_{I|used}$ contains all predicates from $\pi_{I|used}$ that reference relevant variables of the block. The partition $\pi_{I|unused}$ consists of the rest of π_I, like for example predicates never accessed during the block's traversal.

The operator $reduce_{pred}$ removes the set $\pi_{I|unused}$ of unimportant predicates from the precision π_I and from the abstraction state $e_I = (\psi_I, true)$, where the predicates of $\pi_{I|unused}(l)$ are existentially quantified in the abstraction formula ψ_I (where l is the program location of e_I). The operator $expand_{pred}$ re-adds the removed predicates $\pi_{I|unused}$ to the precision π_o and the output state $e_o = (\psi_o, true)$. Therefore the abstraction formula ψ_o is conjuncted with the rest of the initial abstraction formula ψ_I that only consists of predicates of $\pi_{I|unused}$ and is the result of existential quantification of $\pi_{I|used}(l)$ in ψ_I (where l is the program location of e_I). As the path formula is *true* for the input and output locations of a block due to the abstraction computation, it is not changed through the operators $reduce_{pred}$ and $expand_{pred}$. The formal definition of the operators $reduce_{pred}$ and $expand_{pred}$ is:

$$reduce_{pred}(B, (\psi_I, true), \pi_I) = (\exists \pi_{I|unused}(l) : \psi_I, p_{I|used})$$

$$expand_{pred}((\psi_I, true), \pi_I, B, (\psi_o, true), \pi_o)) = (\exists \pi_{I|used}(l) : \psi_I \wedge \psi_o, p_{I|used} \cup \pi_o)$$

There is a small inaccuracy in this procedure, because the expanded abstraction formula might be weaker than the corresponding formula received through a direct analysis of the block without applying the operators $reduce_{pred}$ and $expand_{pred}$. The

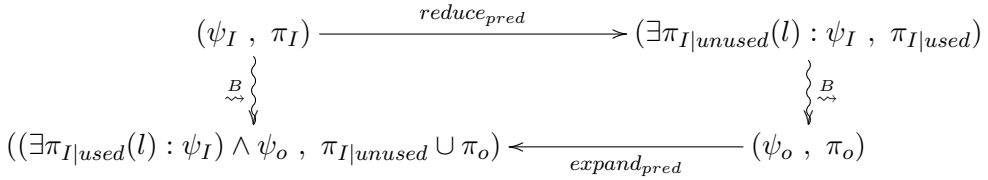


Figure 4.3: Schematic diagram of the operators $reduce_{pred}$ and $expand_{pred}$

soundness of the predicate analysis with BAM is guaranteed, because the expanded abstraction formula over-approximates its counterpart without BAM.

If the operator $expand_{pred}$ is applied to abstract states received from the cache as result of a previous analysis (cache hit), the SSA-indices of the path formula might be wrong for the current program location. As consistent SSA-indices over a sequence of several abstraction states are only important for the counterexample check in the refinements, the path formulae (and their SSA-indices) are re-constructed for these cases to avoid wrong indices from cache hits with BAM.

Figure 4.3 omits the path formulae, as they are always *true* for the abstraction states and thus unimportant for the operators $reduce_{pred}$ and $expand_{pred}$.

4.5.4 Refinement and Interpolation for Recursive Procedures

Analyzing recursive procedures with the predicate analysis is complex and needed several changes in the default control flow. The main problem is based on colliding identifiers in formulae, which are used and produced during the refinement step of CEGAR. After a short description of the changes to the SSA-indices, this section considers the problem of the predicate analysis with colliding identifiers in recursive procedures. Then the solution is explained, which replaces the default sequential interpolation procedure with tree interpolation.

The extraction of predicates from interpolants and the boolean abstraction computation remain unchanged for the analysis of recursive procedures and their execution is equal to the default predicate analysis. Only the interpolation strategy and parts of the formula encoding are modified to allow tree interpolation of found (infeasible) counterexamples.

Parameters for Global and Local Identifiers

To analyze recursive procedures, all function scopes are divided into completely distinct parts. The default predicate analysis only uses an extra identifier for the return value of a function. Additional identifiers for parameters are introduced for formulae to separate the data flow between functions. Also an extra parameter and an extra return variable is added to all function entries and exits for each global variable such that they effectively become local variables.

The proceeding assigns the evaluated arguments of a function call to temporary identifiers that are later assigned to the formal parameters of the function call. The first assignment happens before analyzing the function's entry location, the second one afterwards. Similarly, the assignment of the return variables is done before the function exit location. The return variables are assigned to their target variables after leaving the called function.

The extra variables are encoded transparently while building path formulae. The overhead caused by these additional identifiers is small, because they only appear at function entries and exits and the path formulae only contain more equivalences than before.

SSA for Counterexample

In the context of BAM the path formulae for the counterexample must be recomputed, because the SSA-indices from cached blocks might be invalid. As BAM already provides the counterexample as separate data structure, the path formulae can be constructed by a visitation of its edges. In case of recursive procedures the SSA-indices must be updated after returning from a (recursive) function call, because the called function might have changed the index of an identifier that shadows an equal-named variable from the calling function. Therefore the implementation of the SSA-indices was modified to allow arbitrary increments of indices.

Problem with (Sequential) Interpolation: Collision of Identifiers in the Abstractions

As already mentioned, the predicate analysis computes abstraction formulae at specific program locations. The computation itself uses SSA-indices to distinguish identifiers, even if they represent the same variables. A abstraction formula has no SSA-indices because of its usage for further processing, for example to check coverage between abstract states.

With recursive procedures the analysis has to manage equal-named identifiers from different function scopes, which might be nested and hiding variables from other scopes. The differentiation of identifiers with identical names is only possible as long as SSA-indices are used. Thus in abstraction formulae equal-named variables can not be distinguished any more, i. e. a contradicting predicate like $a < a$ from a SSA-indexed term $a_2 < a_3$ might be used in an abstraction computation. Unsatisfiable or too weak abstraction formulae may lead to either an unsound analysis or an endless iteration in CEGAR or the fixpoint algorithm in BAM.

The problem is caused by sequential interpolation, because this interpolation strategy does not consider scopes of functions. An interpolant (and thus the extracted predicates and the abstraction formulae) might contain the same identifier with different SSA-indices, because all of those indices are valid at the current program location, even if only one index belongs to an identifier that can be accessed in to the current scope and all other indices are out of scope.

Tree Interpolation as Solution for the Collision of Identifiers

The problem of colliding identifiers in abstraction formulae can be circumvented with a better interpolation strategy, where function scopes are considered such that at most one SSA-index for an identifier appears in each interpolant. Tree interpolation guarantees this if the tree of formulae is build according to the control flow of the analyzed program. This section only shows the alignment of formulae in the tree and does not include the dependencies towards BAM-specific components like blocks or the operator *rebuild*, which are considered later.

Interpolation is done as part of the refinement of an infeasible counterexample. Thus the underlying data structure is a (unsatisfiable) sequence of path formulae along the counterexample, where each path formula reaches from one abstraction location to the next one. Abstraction locations are chosen according to function calls such that three abstraction points are present for each function execution: the function call, the function entry and the function return. Those three program location are marked as abstraction points and the interpolation strategy has to provide interpolants for them. The sequence of path formulae of the counterexample will be transformed into a tree of formulae to get a tree interpolation problem instead of sequential interpolation.

Therefore it is important to distinguish two types of function calls along the counterexample. The first type of function calls belongs to function entered and left on the way to the target state, i. e. the function return corresponding to a function

call is executed. The other type consists of functions that are only entered, but never left. This happens if the target state is reached inside a called function. The difference for the analysis is that after leaving a function and further executing the calling function all of the caller's variables are in scope again. If a function is never left, all identifiers from surrounding scopes (except global variables) are invalid in this function. The difference between these two kinds of functions was already considered by Heizmann, Hoenicke and Podelski [17], as their algorithm for nested interpolation also uses distinct steps for both types. With the differentiation of function calls, the building of the tree of formulae can be formulated.

Algorithm 13 gives an overview of the control flow of the interpolation procedure in the refinement: At first a tree of states is build with Algorithm 14 from the abstraction states of the counterexample path depending on the control flow. Then the SMT solver is called to compute tree interpolants. Instead of the SMT solver also similar routines can be executed that return valid tree interpolants, e. g. algorithms for nested or well-scoped interpolation. As the resulting interpolants also form a tree-like structure, Algorithm 15 re-arranges them to a linear sequence of abstraction states to match the abstraction locations along the error path. In the following the mentioned algorithms are explained in detail.

As the abstraction states in the counterexample are unique, the algorithms directly use them as tree-nodes. The labeling $L : V \to Formula$ for the interpolation problem is implicitly given by the abstraction states along the counterexample, because for each of them there is a path formula that corresponds to the path from the abstraction state to the next one. The path formulae for the labeling are taken from the counterexample, because a path formula φ from an abstraction state $e = (\psi, \varphi)$ is always *true*. The following algorithms use several functions:

- $isMainEntry(s)$ returns whether the current abstraction state s is located at the initial program location l_0. This check is important, because the main function is never left, i. e. there is no corresponding function exit along the counterexample.
- $isFunctionEntryWithExit(s)$ returns whether the current abstraction state s is located at the entry of a function that is left via an exit location.
- $isFunctionExit(s)$ returns whether the current abstraction state s is located at the exit location of a function.

Algorithm 14 builds the complete tree from several (temporary) subtrees. Each of the subtrees represents an executed function that appears in the counterexample

Algorithm 13: $getInterpolantsForPath(L, states)$

Input : a list $states$ of pairwise distinct abstraction states,
a labeling $L : states \to Formula$
that assigns a (path) formula to each abstraction state
Output : a list $flattened$ of formulae based on the interpolants of the tree
Variables: a tree $T = (states, E)$ with edges $E \subseteq states \times states$ and a root node $v_r \in states$, a labeling $I : states \to Formula$
that assigns a tree interpolant to each abstract state

```
1 T, v_r := buildTree(states)
2 I := computeTreeInterpolants(T, v_r, L) // ← computed with SMT solver
3 flattened := flattenTree(I, states)
4 return flattened
```

Algorithm 14: $buildTree(states)$

Input : a list $states$ of pairwise distinct abstraction states
Output : a tree $T = (states, E)$ with a root-node $v_r \in states$,
where $E \subseteq states \times states$ denotes the set of edges,
which are directed from the child node to the parent node
Variables: a stack $stack \subseteq states$ of abstraction states

```
1  E := ∅
2  stack = ∅
3  foreach s ∈ states do
4      if isMainEntry(s) ∨ isFunctionEntryWithExit(s) then
           // new leaf, has no children
5          stack.push(s)
6          break
7      else if isFunctionExit(s) then
           // first connect the subtree with the exit location
8          E := E ∪ {(stack.pop(), s)}
           // then add it to the calling function
9          E := E ∪ {(stack.pop(), s)}
10         stack.push(s)
11         break
12     else
           // just create a new edge and switch the top element
13         E := E ∪ {(stack.pop(), s)}
14         stack.push(s)
15         break
16 v_r := stack.pop()
17 T := (states, E)
18 return T, v_r
```

Algorithm 15: $flattenTree(I, states)$

Input : a list *states* of pairwise distinct abstraction states,
a labeling $I : states \to Formula$ as solution of a tree interpolation problem
Output : a list $flattened$ of formulae
Variables: a stack *stack* of formulae

```
1  flattened := ∅
2  stack := ∅
3  foreach s ∈ states do
4      if isMainEntry(s) ∨ isFunctionEntryWithExit(s) then
           // function entry, start new scope
5          flattened.append(true)
6          stack.push(I(s))
7          break
8      else if isFunctionExit(s) then
           // merge function summary and function execution
9          flattened.append(rebuildITP(I(s), stack.pop()))
10         break
11     else
           // add the interpolant to the sequence
12         flattened.append(I(s))
13         break
14 return flattened
```

Algorithm 16: $rebuildITP(x, y)$

Input : two boolean formulae x and y
Output : one boolean formula containing all predicates from x and y

$$\textbf{return} \begin{cases} x, & \text{if } y = true \\ y, & \text{if } x = true \\ x \vee y, & \text{otherwise} \end{cases}$$

and is left via its function exit location. The stack *stack* consists of the root nodes of currently available subtrees. The algorithm iterates over all abstraction states and chooses further steps depending on the control flow at the current program location. For each function entry with a corresponding function return a new subtree is started, i. e. a new leaf is created from the abstraction state at the function's entry location and each abstract state of the function's body is appended as parent of the current subtree. At a function exit the subtree of the called function is connected with the subtree of the calling function. Thus the function exit location has two children: the first one represents the trace of the calling function and the second child is the root of the called function's subtree. Due to the iteration order the called function is already handled completely, when the function exit is reached. If a function in the counterexample has no function exit, the target location of the current counterexample was found during the analysis of this function and its abstraction states are just appended as parents of the current subtree.

The resulting tree $T = (V, E)$ with the root node $v_r \in V$ consists of nodes V of abstraction states and edges $E \subseteq V \times V$ between them. The tree is the input for the SMT solver that computes a solution for the tree interpolation problem. Instead of the direct application of the SMT solver, also similar routines that return tree interpolants can be executed, e. g. an algorithm for well-scoped interpolation (Algorithm 6) or tree interpolation via binary interpolation (Algorithm 7). The resulting labeling $I : V \to Formula$ allows to get an interpolant $I(v)$ for every node $v \in V$ of the tree, where v is represented by an unique abstraction state from the counterexample.

The tree-like structure of interpolants is flattened such that it matches the sequence of abstraction points along the counterexample. Algorithm 15 transforms the tree of formulae into a sequence that is used as input for further steps in the predicate analysis, i. e. the interpolant can be split into predicates and utilized for abstraction computations in the next iteration of the CEGAR-loop. Every function call (with a corresponding function exit) starts with an interpolant $true$, which does not consist of any useful predicate. At function exits Algorithm 16 merges two interpolants into one boolean formula such that all predicates from both interpolants are used afterwards and can be extracted from the merged formula. Algorithm 16 filters out formulae without predicates through the elimination of formulae like $true$ and $false$ and the disjunction of the rest.

Predicate Abstraction at Function Entries vs BAM Cache Hit Rate

Due to Algorithm 15 the reader might expect that the refinement procedure with tree interpolation always yields the interpolant *true* for function entry locations. However this assumption is invalid, because it only considers function calls with a corresponding function exit. If a spurious target state was found in a (maybe recursive) function, the corresponding infeasible counterexample never leaves this function and thus the tree interpolant for the corresponding function entry might differ from *true*.

In the case of a recursive function (or several nested functions) without any inner target state, for example if the target state is part of the (non-recursive) main function, the interpolant (and thus also the abstraction formula) for the function entry is indeed guaranteed to be *true*. Such a situation causes a high hit rate for the cache, because each function block is identified in the cache by its (reduced) abstraction formula *true* of the function entry state (and the corresponding precision).

4.5.5 Rebuild

The operator $rebuild_{pred}$ is based on the function call state $e_{call} = (\psi_{call}, true)$, the (not yet reduced) function entry state $e_I = (\psi_I, true)$, and the expanded function return state $e_O = (\psi_O, true)$. Due to the chosen abstraction locations, the path formulae of those abstract states are *true*, which simplifies further steps.

The path formula φ_{call} of the CFA edge that is between the program locations of the abstract states e_{call} and e_I is the conjunction of all assignments of the arguments to the parameter variables (including additional parameters for global variables). The rebuilding changes the abstraction formula appearing ψ_O at function exit, because ψ_O might contain of variables that are out of scope at the function exit location. If all SSA-related parts is omitted, the operator $rebuild_{pred}$ is defined as

$$rebuild_{pred}(B, e_{call}, e_I, e_O, \pi_O) = \Big((\psi_{call} \wedge \varphi_{call} \wedge \psi_O)^{\pi(l)}, \pi_O \Big).$$

The SSA-indices for all variables are updated with correct values, because the variables (i. e. their equal-named counterparts) might have been used and overridden in the calling function's scope. Parameters and return variables are updated with their latest SSA-index that is used in the called function. The precision π_O remains unchanged.

Abstracting Twice at One Program Location

BAM allows to compute predicate abstraction twice at the same program location, if it is the output location of a block. The two abstractions are done in different analyses of BAM: The first abstraction is computed as (last) part of the block's analysis and results in the abstraction formula ψ_o of the abstract state $e_o = (\psi_o, true)$. The abstraction ψ_o (and in many cases also its expanded form ψ_O) only contains identifiers corresponding to parameters or return variables of the called procedure and summarized the function's execution. The second abstraction is computed with the operator *rebuild* in the analysis of the enclosing block of BAM, when the transfer relation $\rightsquigarrow_{BAM}$ analyzed the step from the initial abstract state $e_I = (\psi_I, true)$ to the rebuild abstract state $e_R = (\psi_R, true)$. At the function exit location l the function call's abstraction ψ_{call}, the parameter assignment φ_{call}, and the summarized function's abstraction ψ_O are used to compute a new abstraction $\psi_R = (\psi_{call} \wedge \varphi_{call} \wedge \psi_O)^{\pi(l)}$ that represents the abstract state in the calling function including the called function's result.

If we ignore the application of the operator *expand*, which might change the set $\pi(l)$ of available predicates at the function exit location l between the two abstraction computations, both abstraction formulae ψ_R and ψ_o are build from the same set $\pi(l)$ of predicates, which is extracted from the combination of boolean formulae during the flattening of the tree interpolants with Algorithm 15.

Proof of Soundness and Progress for the Predicate Analysis for Recursive Procedures

This proof extends the previous proof (see Section 4.5.2) for the default predicate analysis, which only uses sequential interpolation. Proofing the correctness of the predicate analysis for BAM especially with using the operator *rebuild* is more complex, as it uses tree interpolation. For recursive procedures the control flow must be taken into account. As the operator *rebuild* is only applied at function exits, these program locations are specifically considered here.

As mentioned before the soundness of the predicate analysis relies on inductive invariants. For all program location except those function exits, where the operator *rebuild* is applied, the input for the abstraction computation is the previous abstraction formula ψ and the path formula φ representing the path from the last abstraction location to the current one. At function exits the operator *rebuild* uses three formulae as input for the abstraction computation: two abstraction formulae ψ_{call} and ψ_{exit} that correspond to the pre-condition of the called function

and the function summary, and a path formula φ_{call} consisting of the parameter assignments at the function entry. Due to the boolean predicate abstraction the resulting abstraction formula ψ' is implied by the conjunction of given formulae, i. e. $\psi_{call} \wedge \varphi_{call} \wedge \psi_O \Rightarrow \psi'$. Thus there is no exclusion of a possible feasible counterexample during further exploration of the state space.

The progress of CEGAR, i. e. no repetition of counterexamples, is guaranteed by abstraction formulae that are strong enough to exclude all found counterexamples from re-exploration. Each boolean predicate abstraction at a program location l (except function exits) causes an abstraction formula ψ from predicates $\pi(l)$ implying the corresponding interpolant ψ_{Itp}, thus $\psi \Rightarrow \psi_{Itp}$ is satisfied. For each function exit two interpolants $\psi_{ItpExit}$ and ψ_{ItpR} exist, which are both computed in one refinement from one counterexample, but for different positions in the tree interpolation problem. The first interpolant $\psi_{ItpExit}$ is computed for the root-node c of a subtree and the second interpolant ψ_{ItpR} is the label $I(v)$ for the corresponding merge node v, which is the parent node of the node c in the interpolation tree. The two interpolants are merged into one boolean formula by Algorithm 16. Thus there are two valid implications $\psi_{exit} \Rightarrow \psi_{ItpExit}$ and $\psi_R \Rightarrow \psi_{ItpR}$, with a boolean formula ψ_{exit} as the abstraction formula retrieved after computing the last transfer relation inside the function block and a formula ψ_R as the result of the predicate abstraction of the operator *rebuild*. The operator *rebuild* computes the abstraction formula $\psi_R = (\psi_{call} \wedge \varphi_{call} \wedge \psi_O)^{\pi(l)}$, which is the strongest post condition of the function call and the function execution, i. e. after executing the function call and the whole body of the function.

Due to the properties of tree interpolants each interpolant ψ_{Itp} (and thus also the corresponding abstraction formula ψ) is sufficient, if it is combined with other formulas according to the tree of formulae (i. e. matching the control flow), to make the rest of the previously found error path infeasible and the same counterexample is not re-explored in further CEGAR-iterations.

Rebuilding for All Procedure Calls

This thesis only analyzed the usage of tree interpolation for recursive procedures, but there is no necessity for recursion. The interpolation strategy also works for all function calls. As future work the benefit of such an interpolation strategy could be considered to get a modular analysis, because the abstraction formulae received at function exits summarize those functions and might be useful in contexts beyond BAM.

4.5.6 Example for Counterexample with Tree Interpolation and Flattening

The following example describes the refinement procedure for a specific counterexample that might be reached during the analysis of the recursive program in Figure 4.4 with BAM combined with the predicate analysis. Figure 4.5 provides such a counterexample encoded as path formulae along the target path, which unrolls the recursive procedure twice. Only abstraction states are shown and each function scope (i. e. *main*, the first and second unrolling of f) corresponds to another column of nodes in Figure 4.5. Only the function call, function entry, and function exit are necessary for the analysis, but abstractions are also computed at the function return to get a well-formed example.

Under the assumption that the predicate analysis is configured to use LBE, several edges of the CFA are combined into one path formula. The conjunction of all path formulae is unsatisfiable and thus the given counterexample is infeasible. Function summaries are represented as dashed lines and are no part of error path. In contrast to path formulae function summaries do not contain SSA-indices and are only given as hint for the reader. While the variables a, b, x, and tmp belong to the program, the identifiers p and r correspond to additional variables for parameters and return values of function calls. As the identifiers are unique above all functions of the example program, the plain name is used in the formula.

```
1  void main() {
2    int a = 2;
3    int b = f(a);
4    if (b != 2) {
5      error();
6    }
7  }
8
9  int f(int x) {
10   if (x <= 0) {
11     return x;
12   } else {
13     int tmp = f(x - 1);
14     return tmp + 1;
15   }
16 }
```

Figure 4.4: Simple program with recursion (equal to Figure 1.2)

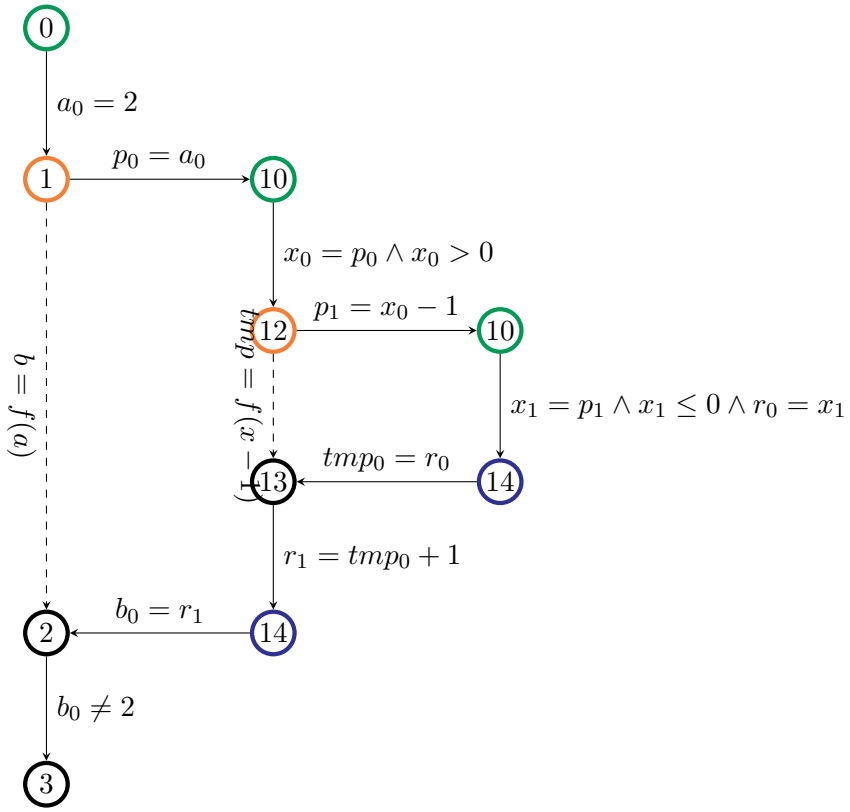


Figure 4.5: Infeasible counterexample of the recursive program in Figure 4.4 with path formulae along the edges, each node is labelled with its program location

In Figure 4.6 the path formulae of the counterexample are arranged in form of a tree that is the result of Algorithm 14 and is the input for the tree interpolation problem. Each node v of the tree is labeled with its formula $L(v)$. Each function call corresponds to a new subtree. The parameter assignment of a function call is the label of the merging node, which is the connection between subtrees of the function's execution and the surrounding scope. Below each node v the computed interpolant $I(v)$ is written in the tree.

Figure 4.7 contains all tree interpolants flattened to match the control flow of the given counterexample. Each function entry corresponds to a new formula *true* that is introduced with Algorithm 15. At each function exit two interpolant are combined into one formula by Algorithm 16. The extraction of predicates is done by the predicate analysis and not shown in this example. The last abstraction state in the counterexample does not have a matching interpolant.

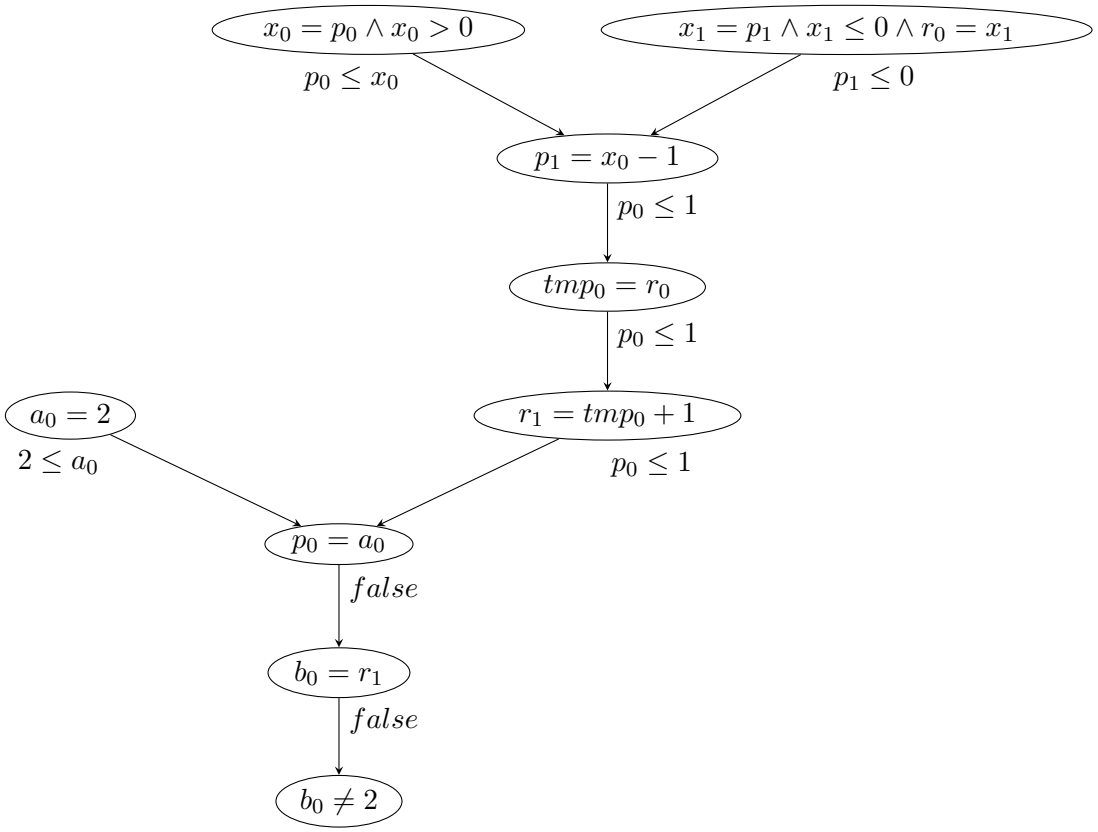


Figure 4.6: Tree of (path) formulae with resulting tree interpolants, each node is labelled with its formula, the interpolants are shown below the nodes

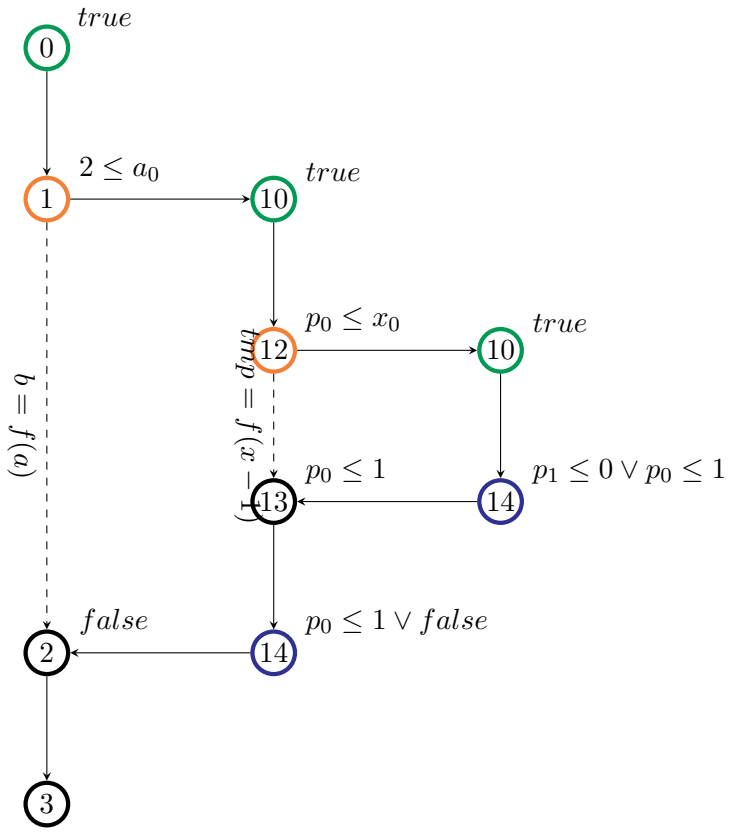


Figure 4.7: Tree interpolants flattened to match the control flow of the counterexample, each node is labelled with its program location and its (rebuild) interpolant

5 Implementation

This chapter describes the implementation in CPAchecker that was done as part of this thesis. As the development of CPAchecker uses a subversion repository, the development took place in a separate branch `bamRecursion`. Parts of the code were already integrated into the main branch `trunk` as bug fixes or as preparation for the International Competition on Software Verification 2015. This section provides an overview of the parts that were modified and extended for the analysis of recursive procedures with BAM.

5.1 BAM-CPA

BAM had been implemented as BAM-CPA within the framework CPAchecker before this thesis and was already used as analysis in the International Competition on Software Verification 2012 [28]. The code of the BAM-CPA, which is available in the package `org.sosy_lab.cpachecker.cpa.bam`, was refactored and last parts of the predicate analysis were removed from the BAM-CPA to get a clean code base.

During the preparation of this thesis, the fixpoint algorithm in the transfer relation of BAM, special handling of function blocks, and the operator *rebuild* were added to the BAM-CPA in order to analyze recursive procedures. It is possible to disable the added components and execute BAM as before by setting the option `cpa.bam.handleRecursiveProcedures=false`. The export the CFA and all ARGs for an execution of BAM was introduced such that blocks and their ARGs are highlighted and connected according the application of the operators *reduce* and *expand*. The output format is DOT [1], which is a simple language to describe graph-like structures. The graphs were mainly used for debugging problems in BAM and its components.

[1] `http://www.graphviz.org/doc/info/lang.html` - last check: March 5, 2015

5.2 Changes in CPAs

The operator *rebuild* is defined in the interface of BAM. The implementation was done for several analyses including ARG-CPA, CallstackCPA, CompositeCPA, LocationCPA, PredicateCPA, and ValueCPA. Here only the two most important implementations of the operator *rebuild* for the PredicateCPA and the ValueCPA are explained, because for the other CPAs the operator returns the identity that was previously defined with $rebuild_{id}$.

The package of the ValueCPA is `org.sosy_lab.cpachecker.cpa.value`. The implementation of $rebuild_{val}$ (defined in Section 4.4.3) in the `ValueAnalysisReducer` was straight forward, because updating an assignment that is stored in an abstract value state is a simple operation on a map. For the refinement in the value analysis the call stack information for the counterexample had to be accessible and was added for several components including `ValueAnalysisFeasibilityChecker` and `ValueAnalysisEdgeInterpolator`.

The PredicateCPA is located in `org.sosy_lab.cpachecker.cpa.predicate`. The operator $rebuild_{pred}$ (defined in Section 4.5.5) was added to `BAMPredicateReducer`. The refinement procedure (described in Section 4.5.4) had to support recursive counterexample paths and tree interpolation. The path formulae for the counterexample are build with the `BAMPredicateRefiner`, where support for recursion was introduced.

Several utilities to build, modify and check formulae (including the integration of SMT solvers) can be found in `org.sosy_lab.cpachecker.util.predicates`. The interpolation strategies were added in the `InterpolationManager`. Therefore the class `CtoFormulaConverter` encodes global variables in path formulae with additional identifiers that are introduced at each function entry and exit according to the description in Section 4.5.4. The reason for the encoding procedure are some combination of interpolation strategies with SMT solvers, because equal identifiers (with equal SSA-indices) in several subtrees of a tree interpolation problem may cause problems.

5.3 Interpolation Strategies

Several strategies for interpolation were implemented in the `InterpolationManager` for this thesis. Beyond the existing strategies for binary and sequential interpolation, CPAchecker is now able to use tree interpolation, nested and well-scoped interpolation with various SMT solvers. The strategies themselves are implemented

in an abstract manner that is (nearly) independent of a specific SMT solver. As already shown in Table 2.1, not all solvers support all interpolation strategies directly. However all available combinations of methods and solvers were implemented and are also compared in Section 6.

The interpolation-related part of the API in SMTInterpol was used as main reference, because it uses simple data structures, which can be reused in CPAchecker. SMTInterpol needs only two arrays to represent a tree of formulae: the first array contains the post-order sorted formulae of the tree, the second one contains plain integer numbers, where each number is the index of the left-most node in the subtree that belongs to the current array-element. The integration of tree interpolation directly via the solver's API was done for SMTInterpol and Z3 in the classes `SmtInterpolInterpolatingProver` and `Z3InterpolatingProver`, respectively. In contrast to SMTInterpol, there was more effort to get tree-interpolation working with Z3, because its API uses a special operator to build a tree of formulae and the integration of Z3 into CPAchecker is not as stable as it could be.

5.4 Configuration of the Analyses

This section provides the necessary configuration options to analyze recursive procedures with the predicate analysis and the value analysis. Therefore the following options are set in CPAchecker as basis for all analyses that handle recursive function calls:

- `cpa.bam.aggressiveCaching=false` enables the usage of the precision as part of the key to access the cache in BAM.
- `cpa.bam.blockHeuristic=FunctionAndLoopPartitioning` configures the heuristic to choose functions and loops as block size for BAM.
- `cpa.bam.handleRecursiveProcedures=true` enables the usage of the fixpoint algorithm and operator *rebuild* in BAM.
- `cpa.callstack.depth=1` allows to analyze recursive procedures in the CallstackCPA. The value 1 is sufficient, because BAM applies the operator *reduce* to the call stack at every function entry.

5.4.1 Value Analysis

The value analysis does not need special options. The used property file is named `valueAnalysis-bam-rec.properties`. It configures BAM with the value analysis and uses the refinement strategy described in Section 4.4.4.

5.4.2 Predicate Analysis

The implementation of BAM and the predicate analysis work with ABE such that both SBE and LBE would be available. The choice of LBE as default configuration has its reasons in the smaller number of formulae per program than with SBE.

The file `predicateAnalysis-bam-rec-plain.properties` was used to set the basic properties. It configures the set of used CPAs (including BAM-CPA and Predicate-CPA) and enables the handling of recursive functions for BAM. Following options configure the abstraction computations and the refinement procedure:

- `cpa.predicate.blk.alwaysAtFunctions=true` and `cpa.predicate.blk.alwaysAtFunctionCallNodes=true` choose the abstraction locations according to function blocks.
- `cpa.predicate.refinement.strategy` chooses the strategy for interpolation and can be either `TREE`, `TREE_WELLSCOPED`, `TREE_NESTED`, or `TREE_CPACHECKER` corresponding to the defined strategies in Section 2.6.3.
- `cpa.predicate.solver` chooses the SMT solver that is used for the analysis. Depending on the configured interpolation strategy the solvers MathSat, Princess, SMTInterpol, and Z3 are available.
- `cpa.predicate.useParameterVariables=true` and `cpa.predicate.useParameterVariablesForGlobals=true` enable the encoding of additional identifiers in formulae.

In the evaluation (Section 6) different combinations of interpolation strategies and SMT solvers are compared. Therefore additionally following option is set:

- `cpa.predicate.encodeBitvectorAs=INTEGER` and `cpa.predicate.encodeFloatAs=INTEGER` set the type of encoded variables to `INTEGER`, which is commonly supported by all SMT solvers and sufficient for the program files of the evaluated benchmark set.

6 Evaluation

In this chapter, the implementation of BAM to analyze recursive procedures in CPAchecker is evaluated. After a description of the benchmarking environment, conditions, and source files, different configurations of CPAchecker are analyzed. Then the implementation of BAM in CPAchecker is compared with other tools that have support for recursive procedures. In the following the token task always denotes an execution of a tool, that tries to verify a program.

6.1 Benchmarks and Source Files

The benchmarks consist of 49 recursive programs generated for this thesis and 24 source files taken from the category of recursive programs from the benchmark set of the International Competition on Software Verification 2015[1]. All 73 source files only consist of simple calculations over integers. The only data type used is the default *int* of the C programming language. There is no usage of data structures like pointer of structs. The source files are intended to be free of possible integer overflows and bit vector computations like binary operations & or |. The property to be verified is the reachability of a function named `__VERIFIER_error()`. If this function can never be called during the program's execution, the program is considered as Safe.

6.2 Resources, Limitations and Measurements

The verification tasks are executed on dedicated compute servers with a 3.4 GHz 64-bit Quad Core CPU (Intel i7-2600) and a Linux operating system (Ubuntu 14.04, x86_64, Linux 3.13.0). Each machine has (at least) 16 GB of RAM, of which the evaluated tools can use exactly 15 GB. For each single verification task the run-time limit is 15 min, measured as CPU time. The tools are allowed to use all 8 available CPU cores (4 physical plus 4 hyper-threading cores).

For each task the values for CPU time, wall time and memory consumption are

[1] `http://sv-comp.sosy-lab.org/2015/benchmarks.php` - last check: March 5, 2015

measured. If a tool runs out of limits for a single task, its process is terminated and the corresponding results are withdrawn.

6.3 Configurations of CPAchecker

CPAchecker is used in SVN revision 15572 from the branch `bamRecursion`, which was used during the development of this thesis. The options for CPAchecker are chosen as stated in Section 5.4. In this section different configurations to process recursive programs are compared, which enable different CPAs or strategies to verify programs. For all configurations the options `-noout` (disable all writing of output) and `-heap 12000M` (size of heap for the Java virtual machine) are set.

6.3.1 Value Analysis

The value analysis in CPAchecker is configured as described in Section 5.4.1. The drawback of the value analysis lies in uninitialized program variables, which cause spurious counterexamples and cannot be sufficiently analyzed by this analysis.

6.3.2 Predicate Analysis

The configuration of CPAchecker for the predicate analysis in described in Section 5.4.2. As the runtime of the predicate analysis relies on the interpolation strategy and the used SMT solver, several different combinations are compared here.

In CPAchecker four distinct tree interpolation strategies are implemented: nested interpolation, well-scoped interpolation, tree interpolation directly via the SMT solver's internal implementation, and tree interpolation in CPAchecker with the Algorithm 7. The executed interpolation strategy can be configured with the option `cpa.predicate.refinement.strategy`. In the tables these four approaches are denoted as *nested*, *wellscoped*, *tree*, and *treeCPA*, respectively. Due to the availability of distinct SMT solvers with different properties, the benchmarks were executed for several combinations of interpolation strategies and solvers.

MathSat and Princess lack support for direct tree interpolation. SMTInterpol does not permit using several distinct solver environments with shared symbols, which is needed for nested interpolation and also for CPAchecker's own implementation of tree interpolation. Thus no benchmarks were executed for these combinations. To allow a direct comparison of all available SMT solvers, numeral formulae are encoded as integers. This is valid, because floats or bit vector-related properties are not used in the recursive source files.

	nested	tree	treeCPA	wellscoped
MathSat	75 1,400 s	-	69 190 s	64 550 s
Princess	44 1,300 s	-	41 1,300 s	43 1,100 s
SMTInterpol	-	62 450 s	-	62 460 s
Z3	57 720 s	44 99 s	56 570 s	54 320 s

Table 6.1: Score and runtime of correct results for combinations of SMT solvers and interpolation strategies in the predicate analysis of CPAchecker (max. score is 112)

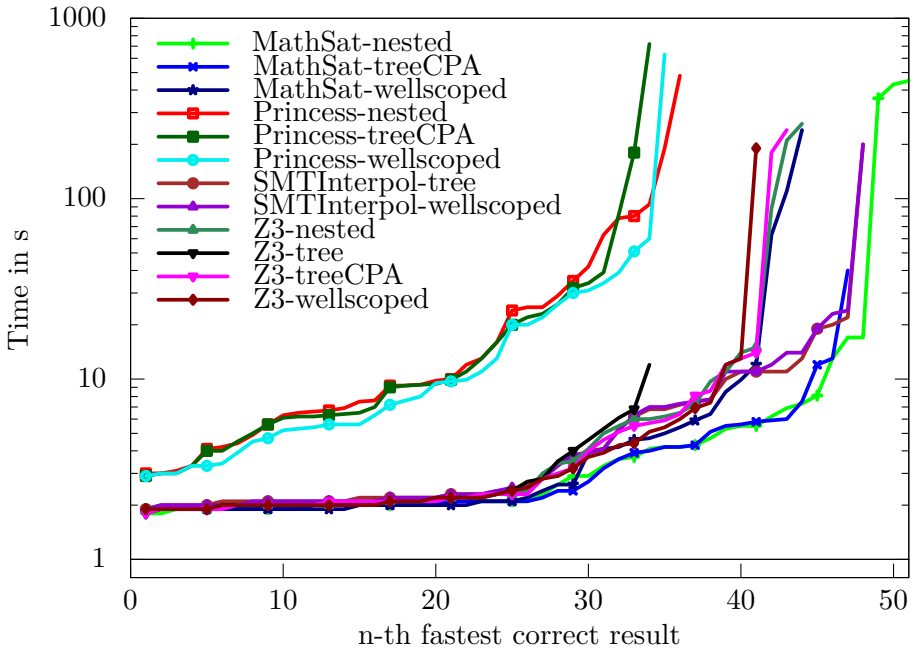


Figure 6.1: Quantile plot for the runtime of correct results of different configurations of the predicate analysis in CPAchecker

Table 6.1 shows the reached score and the runtime of correct results for different combinations of interpolation strategies and SMT solvers. The first entry in each cell of Table 6.1 represents the reached score, the scoring scheme is taken from of the International Competition on Software Verification 2015. The second line contains the runtime (in seconds) of CPAchecker for the correct results only. A smaller runtime does not imply a better strategy if the score is lower. Figure 6.1 shows the quantile plot for the mentioned combinations of interpolation strategies and SMT solvers. The plot contains the successful tasks of each configuration sorted by their runtime.

In Table 6.1 the winner is the combination of nested interpolation with MathSat, which reached a score of 75. This combination was already successfully applied in the International Competition on Software Verification 2015 and reached the fourth place in the category of recursive programs. Detailed results can be looked up in Appendix B.

As all described interpolation strategies are valid to verify programs or confirm counterexamples, differences come from the solvers' implementation and from the predicates extracted from interpolants. Details for different formulae are omitted here, but the overall tendency for the results with different SMT solvers is explained. There are some files that are analyzed instantly and all analyses return correct results for them. Differences between the results are often caused by bugs in BAM that appear in the refinement of the ARGs and cause endless iteration of the CEGAR-loop. Princess is slower than other solvers, because it is a young SMT solver in comparison with the other ones. MathSat is not only a mature program, it also has support for the primitive arithmetic operation *modulo* (with a constant numeric divisor), which appears in one of the source files. All solvers except MathSat find a counterexample in this bug-free source file.

6.4 Configurations of Evaluated Tools

This section provides information about some other software model checkers. All of the evaluated tools participated in the International Competition on Software Verification 2015[2] and scored well in the category of recursive source files. All evaluated tools (except CPAchecker) are configured as for the International Competition on Software Verification 2015 and are expected to be optimized for good results.

The comparison of CPAchecker with other tools is based on two configurations.

[2] `http://sv-comp.sosy-lab.org/2015/participants.php` - last check: March 5, 2015

The first configuration consists of BAM combined with the value analysis, as this is a simple approach and it is expected to find possible bugs fast. The second configuration uses BAM and the predicate analysis with the nested interpolation strategy and MathSat as SMT solver.

6.4.1 CBMC

One of the most known bounded model checkers is CBMC[3], which is used in the version [4] that participated in the International Competition on Software Verification 2015. As bounded model checkers only unwind recursive function up to a certain limit, CBMC is executed with a script that iteratively increments the limit until the program is analyzed completely.

6.4.2 CPArec

CPArec[5] allows to verify recursive C programs via source-to-source program transformation [11]. It needs another program analyzer as underlying verifier and uses function summaries from the verifier's output as inductive invariants to check recursive programs. The current underlying verifier is CPAchecker, which is configured to verify the given program with the predicate analysis.

6.4.3 Smack+Corral

Smack+Corral[6] is a translator from the LLVM compiler's intermediate representation into the Boogie intermediate verification language [27] that is given to either the Boogie or (in this case) the Corral verifier. Because of the usage of LLVM, Smack+Corral is able to analyze several program languages like for example C/C++, Java, Fortran, Erlang and Ruby. Also each optimization available in LLVM can be applied to a program. This includes preprocessing of source code such as constant propagation, dead code elimination, partial loop unwinding and function inlining. The optimized verification problem is given to the verification routine that uses a SMT solver to analyze the program. The tool Smack+Corral scored well in the competition and got the first place for the category of recursive source files.

[3] `http://www.cprover.org/cbmc` - last check: March 5, 2015
[4] `http://www.eecs.qmul.ac.uk/~mt/cbmc-sv-comp-2015.tar.gz` - last check: March 5, 2015
[5] `https://github.com/fmlab-iis/cparec` - last check: March 5, 2015
[6] `http://soarlab.org/research/projects/smack` - last check: March 5, 2015

6.4.4 UltimateAutomizer

The analysis of recursive programs with UltimateAutomizer[7] is based on nested interpolants [17]. Due to the high stage of the tool's development, the approach works better in UltimateAutomizer than its implementation does within CPAchecker. UltimateAutomizer[8] is a twin of UltimateAojak and is based on the same framework. Due to the low score for recursive files in the International Competition on Software Verification 2015 the evaluation of UltimateAojak was omitted here.

6.4.5 Further Tools

The tools LLBMC[9] and ESBMC[10] are bounded model checkers for C programs and use SMT solvers to verify programs. They are able to unwind recursive functions up to a certain limit. As the limit has to be given by the user, an automatic evaluation on a bigger set of programs is not practical.

The verifier Whale that is mentioned in Section 1.3.2 is only a prototype and does not support the specification of the error location. It was impossible to evaluate the tool on the source files of the benchmark.

The framework SeaHorn[11] converts a program into Horn-clauses and solves them by using Z3 as SMT solver [16]. Its predecessor UFO[12] uses an interpolation technique to get invariants for a program's analysis [2]. Both tools do not support the analysis of recursive programs.

6.5 Experimental Results

This section provides information and reasons for the diverse results of the executed tools. Table 6.2 contains statistics for the tools. Beside the reached score and the runtime for all correct results, it lists the numbers of correct results, false positives and false negatives. Figure 6.2 shows the quantile plot for the executed tools, where the results are sorted by their runtime. Detailed data for each executed task is provided in Appendix B.

CBMC does not only limit the number of unwindings for recursive functions to either 6 or 12, depending on the already analyzed parts of the program, but it also

[7] `http://ultimate.informatik.uni-freiburg.de` - last check: March 5, 2015
[8] `https://ultimate.informatik.uni-freiburg.de/Kojak` - last check: March 5, 2015
[9] `http://llbmc.org` - last check: March 5, 2015
[10] `http://www.esbmc.org` - last check: March 5, 2015
[11] `https://bitbucket.org/lememta/seahorn/wiki/Home` - last check: March 5, 2015
[12] `https://bitbucket.org/arieg/ufo/wiki/Home` - last check: March 5, 2015

	CBMC	CPAchecker Predicate	CPAchecker Value	CPArec	Smack+Corral	UltimateAutomizer
Score (max. 112)	2	75	-24	64	18	89
Runtime	10,000 s	1,400 s	660 s	960 s	3,600 s	1,700 s
#Correct Results	59	51	56	45	65	59
#False Positives	0	0	17	0	2	0
#False Negatives	8	0	0	0	6	0

Table 6.2: Statistics for results of the comparison of different tools

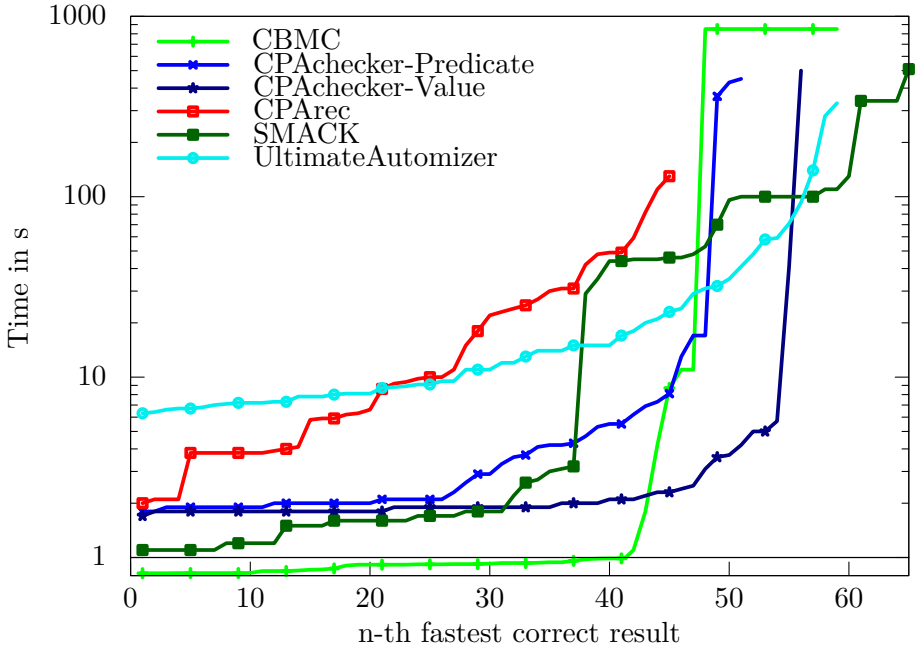


Figure 6.2: Quantile plot for run times of correct results of different tools

limits its runtime to 850 s. If one of the limitations is violated, CBMC aborts its analysis and considers the current source file as SAFE. This results in some false negatives, i. e. some valid counterexamples are not found. The time limitation of CBMC is distinctly visible as horizontal line at 850 s, where the tool terminates and returns a result. On the other side the performance of CBMC can be seen, when the analyzed programs only use initialized variables. Several source files can be proven correct in less than one second of runtime.

SMACK+CORRAL is not able to find bugs in several cases, because the tool attempts to get the minimal number of unrolling loops and recursion by stratified inlining. This method does not work for counterexamples that need deep unwinding of function calls. Thus there are also several false negatives for this verifier.

CPACHECKER, CPAREC and ULTIMATEAUTOMIZER are sound tools and do not return false negatives in the benchmark. CPACHECKER does not return an invalid counterexample, i. e. a false positive, when the predicate analysis is used (with MATHSAT). Due to its lack of handing relations between variables, CPACHECKER finds and returns several spurious counterexample with the value analysis, thus some results are false positive. However the value analysis has a high number of correctly solved verification tasks and allows to analyze even deep recursion within a few second. It is the only analysis that unrolls the source file `id_o1000_false.c` completely, where the recursive function is called 1,000 times, and reports the correct counterexample after a runtime of only 500 s.

7 Conclusion

In this thesis the potential of BAM was explored as a basis for the analysis of recursive procedures. The conceptual contribution is an extension in BAM that allows to handle recursion in an analysis-independent way. The approach was evaluated in several distinct analyses. This chapter concludes this thesis by summarizing the discussed aspects and properties of the extension of BAM to analyze recursive procedures.

7.1 BAM and Recursive Procedures

This thesis contains the first formalization of BAM and its operators in a way that is independent from the underlying analysis. The extension of BAM, which includes the fixpoint algorithm and the operator *rebuild*, to verify recursive procedures is successfully integrated into the framework CPAchecker and is competitive with other state-of-the-art tools.

The formalization and the soundness of the operators *reduce* and *expand* is shown for the predicate and value analysis, and also for other analyses that for example track the program location or the call stack. The operator *rebuild* and the refinement procedure are defined to handle recursion in combination with some CPAs.

7.2 Prospects

The first steps into automatic software verification of (recursive) programs with BAM showed promising results in this thesis. However, there is still work to be done and the analysis can be improved. This section describes some ideas and future work that might be a topic of research and implementation in order to increase the benefit of BAM.

As future work the basic operators of BAM can be defined and implemented for further analyses that for example represent their abstract states with BDDs, intervals, or octagons. Also the combination of different analyses to analyze recursive programs can be beneficial.

7.2.1 Data Structures and Memory Model in BAM

Programs with data types like floating point numerals or bit vectors can already be handled by the current implementation, but the current set of recursive source files only uses integers. The set of source files could be increased to get a better comparison of distinct analyses, tools and their features.

The operators *reduce*, *expand* and also *rebuild* for predicate and value analysis currently only handle variables that are assigned without pointer dereferencing. The current implementation of these operators does not take into account that a program (in the C programming language) might contain pointers and structs, which are used in more complex data structures. The importance of some variable in a block is mainly based on direct access to its identifier. In case of pointer aliasing and complex data structures this is not possible and other methods must be used to distinguish important and unused variables in a block.

7.2.2 Comparison of Interpolation Strategies

There are several distinct interpolation strategies implemented in CPAchecker. Some of them are based on tree interpolation and allow valid refinements for recursive programs. The interpolation strategies should all result in a similar (or maybe even equal) behavior of the predicate analysis. As small discrepancies in interpolants might have a huge effect on the runtime (and termination) of the analysis, the structure of interpolants and extracted predicates is a topic for further research.

7.2.3 Modular Analysis through Predicate Analysis with BAM

If the predicate analysis within the BAM approach is configured to use the operator *rebuild* and tree interpolation, it generates function summaries that consist of only parameter and return variables of the corresponding function call. Such a function summary might be re-used for further analyses of the same program, as it is already done in BAM due to the cache. Additionally the formulae can be applied to other programs that contain the same function call. This modularity might increase the performance of an analysis.

Bibliography

[1] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2012.

[2] Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik. UFO: verification with interpolants and abstract interpretation - (competition contribution). In Piterman and Smolka [26], pages 637–640.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.0*, 2010. Available at `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf` - last check: March 5, 2015.

[4] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 25–32, 2009.

[5] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007.

[6] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Program analysis with dynamic precision adjustment. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 29–38. IEEE, 2008.

[7] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[8] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 189–197. IEEE, 2010.

[9] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7793 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2013.

[10] Régis Blanc, Ashutosh Gupta, Laura Kovács, and Bernhard Kragl. Tree interpolation in vampire. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 173–181. Springer, 2013.

[11] Yu-Fang Chen, Chiao Hsieh, Ming-Hsien Tsai, Bow-Yaw Wang, and Farn Wang. Verifying recursive programs using intraprocedural analyzers. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 118–133. Springer, 2014.

[12] Jürgen Christ and Jochen Hoenicke. *Interpolation in SMTLIB 2.0*, 2012. Available at `http://ultimate.informatik.uni-freiburg.de/smtinterpol/proposal.pdf` - last check: March 5, 2015.

[13] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Proof tree preserving interpolation. In Piterman and Smolka [26], pages 124–138.

[14] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[15] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 331–344. ACM, 2011.

[16] Arie Gurfinkel, Temesghen Kahsai, and Jorge A. Navas. SeaHorn: a framework for verifying c programs - (competition contribution). In *TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, 2015.

[17] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 471–482. ACM, 2010.

[18] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 232–244. ACM, 2004.

[19] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 58–70. ACM, 2002.

[20] Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[21] Charles A. R. Hoare. Procedures and parameters: An axiomatic approach. In Erwin Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, 1971.

[22] Charles A. R. Hoare. The verifying compiler, a grand challenge for computing research. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 78–78. Springer, 2005.

[23] Krystof Hoder, Laura Kovács, and Andrei Voronkov. Playing in the grey area of proofs. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 259–272. ACM, 2012.

[24] Kenneth L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[25] Kenneth L. McMillan. Lazy abstraction with interpolants. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[26] Nir Piterman and Scott A. Smolka, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*. Springer, 2013.

[27] Zvonimir Rakamaric and Michael Emmi. SMACK: decoupling source language details from verifier implementations. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014.

[28] Daniel Wonisch. Block abstraction memoization for CPAchecker - (competition contribution). In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 -*

April 1, 2012. Proceedings, volume 7214 of *Lecture Notes in Computer Science*, pages 531–533. Springer, 2012.

[29] Daniel Wonisch and Heike Wehrheim. Predicate analysis with block-abstraction memoization. In Toshiaki Aoki and Kenji Taguchi, editors, *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, volume 7635 of *Lecture Notes in Computer Science*, pages 332–347. Springer, 2012.

A Tree Interpolation as Extension for SMTLIB Version 2

This section provides useful information about tree interpolation with the SMT solver SMTInterpol and uses the path formulae of an example program that was mentioned before in this thesis (see Figure 4.6). The interpolants in the thesis were simplified to improve readability, but here the exact result of SMTInterpol is given.

```
1  (set-option :print-success false)
2  (set-option :produce-proofs true)
3  (set-option :verbosity 3)
4  (set-logic QF_UFLIA)
5
6  (declare-fun a0 () Int)
7  (declare-fun b0 () Int)
8  (declare-fun p0 () Int)
9  (declare-fun p1 () Int)
10 (declare-fun r0 () Int)
11 (declare-fun r1 () Int)
12 (declare-fun x0 () Int)
13 (declare-fun x1 () Int)
14 (declare-fun tmp0 () Int)
15
16 (assert (! (= a0 2) :named F1))
17 (assert (! (and (= x0 p0) (> x0 0)) :named F2))
18 (assert (! (and (= x1 p1) (<= x1 0) (= r0 x1)) :named F3))
19 (assert (! (= p1 (- x0 1)) :named F4))
20 (assert (! (= tmp0 r0) :named F5))
21 (assert (! (= r1 (+ tmp0 1)) :named F6))
22 (assert (! (= p0 a0) :named F7))
23 (assert (! (= b0 r1) :named F8))
24 (assert (! (distinct b0 2) :named F9))
25
26 (check-sat)
27 (get-interpolants F1 (F2 (F3) F4 F5 F6) F7 F8 F9)
28
29 (exit)
```

Figure A.1: Input for SMTInterpol in SMTLIB2-format (extended for interpolation)

The input and output of the solver is given in Figure A.1 and Figure A.2. The input first configures the solver with some default properties, then all symbols are declared the formulae are pushed onto the solver's stack. Each formula has an unique name (F1-9) to identify them in the tree. The tree interpolation problem is formatted as suggested in the proposal [12], i. e. each right subtree of formulae is written in brackets. The tree structure matches Figure A.3, where the identifiers of the formulae and the corresponding interpolants are shown. The solution for the tree interpolation problem is printed in post order in Figure A.2.

```
unsat
(
    (<= 0 (+ a0 (- 2)))
    (<= p0 x0)
    (<= 0 (- p1))
    (<= p0 1)
    (<= p0 1)
    (<= p0 1)
    false
    false
)
```

Figure A.2: Output of SMTInterpol for the given input

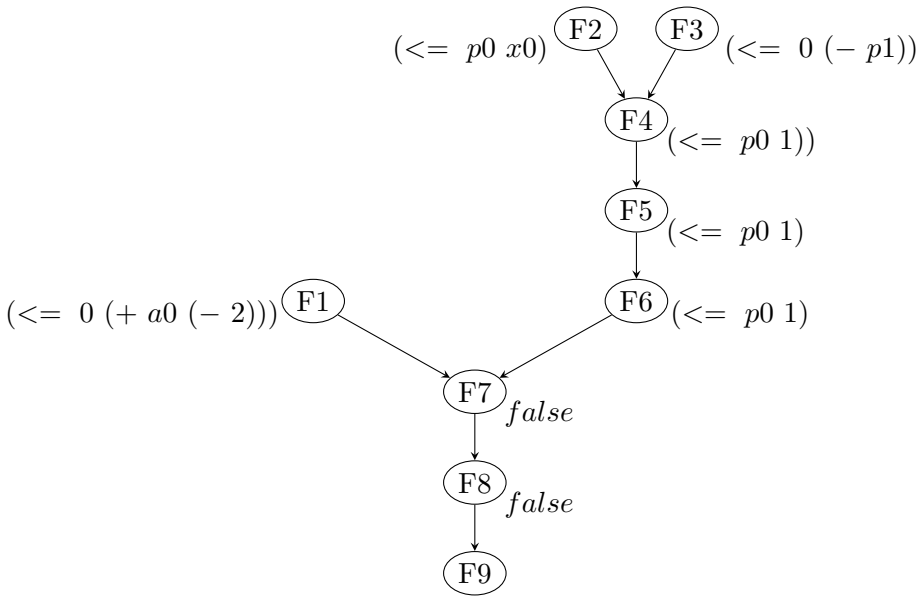


Figure A.3: Tree of formulae with interpolants, each node is labelled with the identifier of its formula, the interpolants are shown near beside the nodes

B Detailed Results of the Evaluation

Table B.1 and Table B.2 show the detailed result and data for each task of the benchmarks. The first table contains the verification result and the measured CPU time for the comparison of different interpolation strategies and SMT solvers, which are evaluated in Section 6.3.2. The second table compares different tools according to Section 6.5. Each line of a table contains the results for one source file, whose name appears in the first column. A source file is considered as SAFE if its name ends with "_T". Otherwise the program ends with "_F" and has a bug, i. e. a feasible counterexample violating the specification. The columns of the tables show the result of the analysis, the needed CPU time (up to a time limit of 900 s) and (only in the second table) the amount of used memory. The score is computed following the scoring scheme of the International Competition on Software Verification 2015.

In the tables we notice that the predicate analysis is not able to analyze recursion, where deep unrolling or many function calls are needed to verify a program. For example the source files `fibo*`, `Fibonacci*`, and `Ackermann` contain recursive implementations of the Fibonacci-sequence or the Ackermann function and need several unrollings of the recursive functions, which is called with distinct input values.

	MathSat			Princess			SMTInterpol		Z3			
	nested	treeCPA	wellscoped	nested	treeCPA	wellscoped	tree	wellscoped	nested	tree	treeCPA	wellscoped
afterrec_2calls_F	✓ 1.9	✓ 1.9	✓ 1.9	✓ 4.4	✓ 4.5	✓ 3.9	✓ 2.0	✓ 2.0	✓ 2.0	✓ 2.0	✓ 1.9	✓ 1.9
afterrec_2calls_T	✓ 1.9	✓ 1.9	✓ 1.9	✓ 4.2	✓ 4.0	✓ 3.3	✓ 2.0	✓ 2.0	✓ 1.9	✓ 2.0	✓ 1.9	✓ 1.9
afterrec_F	✓ 1.9	✓ 1.9	✓ 1.9	✓ 4.9	✓ 5.0	✓ 4.5	✓ 2.1	✓ 2.0	✓ 2.0	✓ 2.0	✓ 2.1	✓ 2.0
afterrec_T	✓ 1.8	✓ 1.9	✓ 1.9	✓ 4.1	✓ 4.0	✓ 3.4	✓ 1.9	✓ 2.0	✓ 1.9	✓ 1.9	✓ 2.1	✓ 2.0
fibo_10_F	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
fibo_10_T	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
fibo_15_F	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
fibo_15_T	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
fibo_2calls_10_F	✓ 430	TO	✓ 110	TO	TO	TO	✓ 19	✓ 24	✓ 210	TO	✓ 180	TO
fibo_2calls_10_T	✓ 450	TO	✓ 63	TO	TO	TO	TO	TO	TO	TO	TO	TO
fibo_2calls_2_F	✓ 2.0	✓ 2.0	✓ 2.2	✓ 6.9	✓ 5.6	✓ 5.2	✓ 2.2	✓ 2.2	✓ 2.1	✓ 2.0	✓ 2.0	✓ 2.4
fibo_2calls_2_T	✓ 4.7	✓ 5.1	✓ 4.7	✓ 29	✓ 20	✓ 30	✓ 4.1	✓ 4.1	✓ 6.0	TO	✓ 5.9	✓ 4.3
fibo_2calls_4_F	✓ 5.3	✓ 5.6	✓ 5.0	✓ 78	TO	TO	✓ 6.2	✓ 6.3	✓ 5.5	TO	✓ 8.1	TO
fibo_2calls_4_T	✓ 8.1	✓ 7.5	✓ 8.5	✓ 63	TO	✓ 39	TO	TO	TO	TO	TO	TO
fibo_2calls_5_F	✓ 5.5	TO	✓ 6.4	TO	✓ 720	✓ 630	✓ 7.7	✓ 7.7	TO	TO	? 5.8	TO
fibo_2calls_5_T	✓ 17	TO	TO	✓ 480	✓ 180	✓ 51	TO	TO	TO	TO	TO	? 3.6
fibo_2calls_6_F	✓ 17	TO	✓ 9.9	TO	TO	TO	✓ 11	✓ 14	✓ 11	TO	✓ 12	TO
fibo_5_F	✓ 3.6	✓ 3.9	TO	✓ 190	TO	TO	✓ 11	✓ 11	✓ 14	TO	TO	TO
fibo_5_T	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
fibo_7_F	✓ 360	✓ 12	TO	TO	TO	TO	✓ 22	✓ 23	TO	TO	TO	TO
fibo_7_T	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
id2_b2_o3_T	✓ 1.9	✓ 2.0	✓ 2.0	✓ 10	✓ 9.3	✓ 11	✓ 2.0	✓ 2.1	✓ 2.0	✓ 2.0	✓ 1.9	✓ 2.0
id2_b3_o2_F	✓ 2.1	✓ 2.1	✓ 2.1	✓ 9.8	✓ 9.9	✓ 9.5	✓ 2.4	✓ 2.4	✓ 2.3	✓ 2.2	✓ 2.3	✓ 2.5
id2_b3_o5_T	✓ 1.9	✓ 2.0	✓ 2.0	✓ 16	✓ 16	✓ 13	✓ 2.1	✓ 2.1	✓ 2.0	✓ 2.0	✓ 2.1	✓ 1.9
id2_b5_o10_T	✓ 2.0	✓ 2.0	✓ 2.0	✓ 24	✓ 22	✓ 20	✓ 2.1	✓ 2.2	✓ 2.0	✓ 2.0	✓ 2.0	✓ 2.0
id2_i5_o5_F	✓ 2.1	✓ 2.1	✓ 2.0	✓ 12	✓ 11	✓ 9.7	✓ 2.4	✓ 2.3	✓ 2.3	✓ 2.2	✓ 2.2	✓ 2.2
id2_i5_o5_T	✓ 4.3	✓ 4.2	✓ 4.1	TO	TO	TO	✓ 6.8	✓ 7.0	✓ 6.0	✓ 5.3	✓ 5.7	✓ 5.4
id_b2_o3_T	✓ 2.0	✓ 1.9	✓ 1.9	TO	TO	TO	✓ 2.1	✓ 2.1	✓ 2.0	✓ 2.1	✓ 2.3	✓ 2.0
id_b3_o2_F	✓ 2.0	✓ 2.1	✓ 2.0	✓ 7.6	✓ 7.0	✓ 6.3	✓ 2.3	✓ 2.2	✓ 2.1	✓ 2.1	✓ 2.1	✓ 2.2
id_b3_o5_T	✓ 1.9	✓ 1.9	✓ 1.9	TO	TO	TO	✓ 2.2	✓ 2.1	✓ 2.1	✓ 2.1	✓ 2.0	✓ 2.1
id_b5_o10_T	✓ 1.9	✓ 1.9	✓ 1.9	TO	TO	TO	✓ 2.1	✓ 2.3	✓ 2.0	✓ 2.0	✓ 2.1	✓ 2.0
id_i10_o10_F	✓ 4.2	✓ 4.2	TO	✓ 25	✓ 23	✓ 20	✓ 11	✓ 11	✓ 5.0	✓ 6.1	✓ 4.6	✓ 6.9
id_i10_o10_T	✓ 5.5	✓ 5.5	TO	TO	TO	TO	✓ 11	✓ 12	✓ 9.7	✓ 6.8	✓ 8.6	✓ 7.4
id_i5_o5_F	✓ 2.6	✓ 2.4	TO	✓ 13	✓ 13	✓ 9.9	✓ 3.7	✓ 3.8	✓ 3.0	✓ 2.8	✓ 3.0	✓ 2.9
id_i5_o5_T	✓ 3.3	✓ 3.2	TO	TO	TO	TO	✓ 5.3	✓ 5.2	✓ 4.1	✓ 3.5	✓ 4.0	✓ 3.9
id_o1000_F	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
id_o100_F	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
id_o10_F	✓ 4.2	✓ 4.3	✓ 4.2	✓ 25	✓ 26	✓ 22	✓ 7.5	✓ 7.5	✓ 6.2	✓ 4.0	✓ 5.1	✓ 4.4
id_o200_F	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
id_o20_F	✓ 13	✓ 13	✓ 12	✓ 80	✓ 82	✓ 60	✓ 20	✓ 19	✓ 15	✓ 12	✓ 14	✓ 13
id_o3_F	✓ 2.1	✓ 2.1	✓ 2.1	✓ 9.3	✓ 9.4	✓ 7.6	✓ 2.3	✓ 2.3	✓ 2.3	✓ 2.3	✓ 2.3	✓ 2.2
sum_10x0_F	✓ 6.2	✓ 5.9	✓ 4.6	✓ 93	✓ 32	✓ 26	✓ 6.8	✓ 7.0	✓ 6.5	✓ 4.6	✓ 5.5	✓ 5.1
sum_10x0_T	✓ 6.9	✓ 5.8	✓ 5.4	✓ 42	✓ 39	✓ 31	✓ 13	✓ 14	✓ 88	TO	✓ 13	✓ 12
sum_2x3_F	✓ 2.0	✓ 2.0	✓ 1.9	✓ 7.5	✓ 6.5	✓ 5.6	✓ 2.2	✓ 2.1	✓ 2.1	✓ 2.0	✓ 2.0	✓ 2.0
sum_2x3_T	✓ 2.9	✓ 2.4	✓ 2.4	✓ 9.2	✓ 9.0	✓ 7.2	✓ 3.5	✓ 3.3	✓ 3.4	TO	✓ 2.8	✓ 2.8
sum_non_eq_F	✓ 1.9	✓ 1.9	✓ 1.9	✓ 3.0	✓ 2.9	✓ 3.0	✓ 2.1	✓ 2.0	✓ 1.9	✓ 1.8	✓ 1.9	✓ 1.9
sum_non_eq_T	✓ 2.1	✓ 2.0	✓ 1.9	✓ 6.7	✓ 6.2	✓ 5.6	✓ 2.3	✓ 2.5	✓ 2.2	✓ 2.1	✓ 2.1	✓ 2.1
sum_non_F	✓ 1.8	✓ 1.9	✓ 1.9	✓ 3.0	✓ 3.0	✓ 2.9	✓ 2.2	✓ 1.9	✓ 1.9	✓ 1.9	✓ 1.8	✓ 2.0
sum_non_T	✓ 2.3	✓ 2.1	✓ 2.0	✓ 6.6	✓ 6.4	✓ 5.6	✓ 2.6	✓ 2.5	✓ 2.2	✓ 2.4	✓ 2.2	✓ 2.2
Ackermann02_F	✓ 2.9	✓ 2.7	✓ 2.6	? 10	? 10	? 8.4	✓ 4.0	✓ 3.9	TO	TO	TO	✓ 3.7
Addition02_F	✓ 2.1	✓ 2.0	✓ 2.0	✓ 5.6	✓ 6.3	✓ 5.4	✓ 2.2	✓ 2.2	✓ 2.1	✓ 2.0	✓ 2.1	✓ 2.1
Addition03_F	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
BallRajamani_F	✓ 2.0	✓ 2.0	✓ 2.0	✓ 6.5	✓ 6.1	✓ 4.7	✓ 2.1	✓ 2.1	✓ 2.1	TO	✓ 2.0	✓ 2.0
EvenOdd03_F	✓ 1.9	✓ 1.9	✓ 1.9	✓ 3.3	✓ 3.3	✓ 3.3	✓ 2.1	✓ 2.1	✓ 2.0	✓ 1.9	✓ 1.9	✓ 2.0
Fibonacci04_F	✓ 4.1	✓ 3.6	✓ 3.9	✓ 35	✓ 34	✓ 34	✓ 7.2	✓ 7.3	✓ 7.3	TO	✓ 6.4	✓ 6.0
Fibonacci05_F	TO	✓ 40	✓ 240	TO	TO	TO	✓ 200	✓ 200	✓ 260	TO	✓ 240	✓ 190
McCarthy91_F	✓ 2.0	✓ 1.9	✓ 1.9	✓ 3.1	✓ 3.0	✓ 3.0	✓ 2.0	✓ 2.0	✓ 1.9	✓ 1.9	✓ 2.0	✓ 1.9
Ackermann01_T	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
Ackermann03_T	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
Ackermann04_T	TO	TO	TO	? 10	? 11	? 8.0	TO	TO	TO	TO	TO	TO
Addition01_T	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
EvenOdd01_T	✓ 3.7	✓ 4.0	✓ 2.6	X 3.3	X 3.3	X 3.3	X 2.1	X 2.0	X 1.9	X 1.9	X 1.9	X 1.9
Fibonacci01_T	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
Fibonacci02_T	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
Fibonacci03_T	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
McCarthy91_T	✓ 2.1	✓ 2.1	✓ 2.1	✓ 9.2	✓ 9.2	✓ 8.0	✓ 2.8	✓ 2.8	✓ 2.4	✓ 2.2	✓ 2.3	✓ 2.3
MultCommutative_T	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
Primes_T	TO	TO	TO	TO	? 140	TO	TO	TO	TO	TO	TO	TO
gcd01_T	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
gcd02_T	TO	TO	TO	? 260	? 49	? 45	TO	TO	TO	? 32	TO	? 30
recHanoi01_T	✓ 2.0	✓ 2.2	TO	TO	TO	TO	✓ 2.2	✓ 2.1	TO	✓ 2.7	TO	TO
recHanoi02_T	✓ 2.0	✓ 2.0	✓ 1.9	✓ 6.3	✓ 6.2	✓ 5.3	✓ 2.1	✓ 2.1	✓ 2.1	✓ 2.1	✓ 2.1	✓ 2.0
recHanoi03_T	✓ 7.3	✓ 6.0	✓ 5.9	TO	TO	TO	✓ 10	✓ 11	✓ 3.5	TO	✓ 3.2	✓ 3.2
Score (max. 112)	75	69	64	44	41	43	62	62	57	44	56	54
#Correct Results	51	47	44	36	34	35	48	48	44	34	43	41
#False Positive	0	0	0	1	1	1	1	1	1	1	1	1
#False Negative	0	0	0	0	0	0	0	0	0	0	0	0

Table B.1: Verification result and runtime (CPU time in seconds) for different configurations of CPAchecker (correct result: ✓, wrong result: X, timeout (900 s): TO, unknown: ?)

	CBMC			CPAchecker Predicate			CPAchecker Value			CPArec			Smack+Corral			Ultimate Automizer		
afterrec_2calls_F	✓	0.14	24	✓	1.9	150	✓	1.8	120	✓	9.8	170	✓	1.2	68	✓	7.2	240
afterrec_2calls_T	✓	0.61	24	✓	1.9	160	✓	1.8	140	TO	900	460	✓	1.8	61	✓	6.7	250
afterrec_F	✓	0.14	24	✓	1.9	160	✓	1.8	140	✓	9.4	160	✓	1.1	57	✓	7.1	250
afterrec_T	✓	0.63	24	✓	1.8	160	✓	1.8	120	TO	900	470	✓	1.6	53	✓	6.8	240
fibo_10_F	✓	0.40	26	TO	900	5,900	✓	2.4	160	OOM	860	15,000	✓	48	1,400	✓	58	550
fibo_10_T	✓	0.82	24	TO	900	5,600	✓	4.2	240	OOM	870	15,000	✓	44	1,400	✓	93	480
fibo_15_F	✓	4.2	58	TO	900	6,100	✓	5.0	230	TO	900	5,900	X	45	1,300	?	310	4,300
fibo_15_T	✓	11	28	TO	900	5,900	✓	41	3,400	TO	900	5,800	✓	45	1,400	?	320	4,900
fibo_2calls_10_F	✓	0.29	26	✓	430	290	✓	2.5	160	TO	900	1,200	✓	110	4,900	✓	280	640
fibo_2calls_10_T	✓	0.90	25	✓	450	340	✓	5.0	360	TO	900	1,200	✓	110	4,800	✓	330	690
fibo_2calls_2_F	✓	0.15	24	✓	2.0	160	✓	1.9	130	✓	6.6	170	✓	100	4,900	✓	7.8	250
fibo_2calls_2_T	✓	0.64	24	✓	4.7	220	✓	1.8	140	TO	900	8,600	✓	100	5,600	✓	8.8	270
fibo_2calls_4_F	✓	0.15	32	✓	5.3	220	✓	2.2	140	✓	18	240	✓	100	4,700	✓	12	270
fibo_2calls_4_T	✓	0.63	24	✓	8.1	240	✓	2.0	140	TO	900	2,200	✓	100	4,900	✓	15	290
fibo_2calls_5_F	✓	0.15	24	✓	5.5	170	✓	1.9	150	✓	27	240	✓	100	5,100	✓	17	290
fibo_2calls_5_T	✓	0.64	24	✓	17	330	✓	2.1	150	TO	900	3,500	✓	100	5,800	✓	20	360
fibo_2calls_6_F	?	0.14	24	✓	17	250	✓	2.0	150	✓	42	560	✓	100	4,800	✓	23	360
fibo_5_F	✓	0.26	25	✓	3.6	210	✓	1.9	140	✓	25	390	✓	2.6	130	✓	14	390
fibo_5_T	✓	0.63	25	TO	900	5,600	✓	2.3	130	✓	30	380	✓	45	1,400	✓	15	290
fibo_7_F	✓	0.26	24	✓	360	4,800	✓	2.0	150	✓	130	1,800	✓	3.0	130	✓	24	360
fibo_7_T	✓	0.66	32	TO	900	6,700	✓	3.1	210	TO	900	1,700	✓	44	1,400	✓	29	370
id2_b2_o3_T	✓	0.95	32	✓	1.9	150	✓	2.1	130	✓	5.9	150	✓	1.7	54	✓	8.0	270
id2_b3_o2_F	?	0.14	24	✓	2.1	150	✓	1.9	140	✓	11	170	✓	1.2	59	✓	7.8	270
id2_b3_o5_T	✓	0.95	33	✓	1.9	150	✓	1.9	120	✓	5.8	150	✓	1.7	56	✓	9.5	280
id2_b5_o10_T	✓	0.94	32	✓	2.0	160	✓	2.0	130	✓	5.9	150	✓	1.7	56	✓	14	420
id2_i5_o5_F	?	0.13	24	✓	2.1	130	✓	1.9	140	✓	24	170	✓	1.2	59	✓	9.1	270
id2_i5_o5_T	✓	0.64	24	✓	4.3	230	✓	1.8	140	TO	900	780	✓	1.6	54	✓	11	290
id_b2_o3_T	✓	0.71	27	✓	2.0	130	✓	1.8	120	✓	3.8	150	✓	1.6	52	✓	7.8	280
id_b3_o2_F	?	0.13	24	✓	2.0	160	✓	1.8	140	✓	10	180	✓	1.1	58	✓	8.1	250
id_b3_o5_T	✓	0.70	27	✓	1.9	160	✓	1.9	120	✓	3.8	140	✓	1.6	52	✓	9.1	280
id_b5_o10_T	✓	0.70	29	✓	1.9	150	✓	1.8	140	✓	3.8	140	✓	1.6	54	✓	13	270
id_i10_o10_F	X	0.32	24	✓	4.2	220	✓	1.9	150	✓	48	220	✓	1.8	60	✓	15	270
id_i10_o10_T	✓	0.61	24	✓	5.5	240	✓	1.9	140	✓	59	190	✓	1.5	54	✓	18	280
id_i5_o5_F	X	0.24	24	✓	2.6	160	✓	1.8	140	✓	23	170	✓	1.1	54	✓	8.9	270
id_i5_o5_T	✓	0.61	24	✓	3.3	170	✓	1.9	140	✓	31	160	✓	1.5	52	✓	11	270
id_o1000_F	X	0.69	26	TO	900	890	✓	500	1,500	TO	900	2,200	X	1.5	52	TO	900	430
id_o100_F	X	0.69	27	TO	900	830	✓	3.7	210	TO	900	2,200	X	1.5	52	TO	900	440
id_o10_F	✓	0.33	25	✓	4.2	150	✓	1.9	150	✓	49	210	✓	1.8	61	✓	12	270
id_o200_F	X	0.69	26	TO	900	910	✓	5.7	230	TO	900	2,200	X	1.5	52	TO	900	450
id_o20_F	✓	0.56	25	✓	13	240	✓	2.1	150	✓	110	360	X	1.6	52	✓	35	360
id_o3_F	✓	0.26	24	✓	2.1	160	✓	1.8	140	✓	15	150	✓	1.1	57	✓	7.2	250
sum_10x0_F	✓	0.35	24	✓	6.2	230	✓	1.8	150	✓	49	260	✓	1.8	61	✓	15	270
sum_10x0_T	✓	0.61	24	✓	6.9	240	✓	2.3	160	✓	3.8	150	✓	1.5	52	✓	21	280
sum_2x3_F	✓	0.14	33	✓	2.0	160	✓	1.8	150	✓	10.0	170	✓	1.1	53	✓	7.2	250
sum_2x3_T	✓	0.61	24	✓	2.9	200	✓	1.9	140	✓	3.9	150	✓	1.5	52	✓	8.7	270
sum_non_eq_F	✓	0.15	24	✓	1.9	130	✓	1.8	140	✓	2.1	140	✓	1.2	56	✓	6.4	250
sum_non_eq_T	✓	8.7	39	✓	2.1	160	X	1.7	150	✓	3.8	150	X	1.1	51	TO	900	440
sum_non_F	✓	0.15	24	✓	1.8	150	✓	1.8	140	✓	2.0	160	✓	1.1	54	✓	6.6	240
sum_non_T	✓	11	45	✓	2.3	160	X	1.8	120	✓	3.8	150	X	1.1	54	✓	31	290
Ackermann02_F	✓	1.8	74	✓	2.9	140	✓	1.7	140	✓	22	370	✓	35	1,200	✓	8.1	240
Addition02_F	✓	0.16	25	✓	2.1	130	✓	1.9	130	✓	6.3	170	✓	3.1	170	✓	7.3	240
Addition03_F	X	850	10,000	TO	900	5,200	✓	1.8	140	TO	900	1,700	X	70	2,200	TO	900	470
BallRajamani_F	✓	0.15	24	✓	2.0	160	✓	1.8	140	✓	6.2	170	✓	1.2	59	✓	7.3	250
EvenOdd03_F	?	0.15	33	✓	1.9	160	✓	1.9	120	✓	2.1	170	✓	1.1	53	✓	6.7	250
Fibonacci04_F	X	0.56	33	✓	4.1	140	✓	1.8	140	✓	31	410	✓	2.7	120	✓	14	290
Fibonacci05_F	X	850	230	TO	900	5,600	✓	1.8	140	TO	900	390	✓	3.2	120	✓	48	400
McCarthy91_F	?	0.14	26	✓	2.0	130	✓	1.8	140	✓	2.1	160	✓	2.2	180	✓	6.3	240
Ackermann01_T	✓	850	320	TO	900	1,100	X	1.8	140	✓	4.0	150	✓	340	10,000	✓	11	290
Ackermann03_T	✓	850	320	TO	900	6,600	X	1.8	140	TO	900	1,500	✓	340	10,000	✓	41	290
Ackermann04_T	✓	850	310	TO	900	6,400	X	1.8	140	TO	900	1,600	✓	340	10,000	✓	32	290
Addition01_T	✓	850	9,100	TO	900	4,900	X	1.9	130	✓	9.2	140	✓	70	2,200	TO	900	460
EvenOdd01_T	✓	1.1	33	✓	3.7	180	X	1.7	140	?	2.1	170	✓	1.7	57	TO	900	450
Fibonacci01_T	✓	850	230	TO	900	5,000	X	1.8	140	✓	82	920	✓	46	1,500	✓	140	340
Fibonacci02_T	✓	0.74	24	TO	900	5,700	✓	3.6	220	TO	900	4,300	✓	46	1,600	✓	59	530
Fibonacci03_T	✓	850	230	TO	900	5,700	X	1.8	140	TO	900	380	✓	45	1,500	✓	71	660
McCarthy91_T	✓	850	1,200	✓	2.1	160	X	1.8	140	✓	8.6	170	✓	29	1,100	✓	8.1	250
MultCommutative_T	✓	850	370	TO	900	450	X	1.9	150	?	29	160	✓	130	3,100	TO	900	470
Primes_T	✓	850	250	TO	900	360	X	2.0	120	TO	900	1,100	✓	510	6,000	TO	900	580
gcd01_T	✓	850	3,500	TO	900	4,300	X	1.8	140	✓	4.1	150	✓	96	2,300	✓	9.5	280
gcd02_T	✓	850	220	TO	900	350	X	1.8	150	TO	900	8,900	✓	340	4,100	TO	900	490
recHanoi01_T	✓	850	670	✓	2.0	160	X	1.8	140	TO	900	1,200	✓	53	1,900	?	390	2,900
recHanoi02_T	✓	0.70	27	✓	2.0	150	X	2.4	170	✓	3.8	150	✓	1.6	52	✓	7.0	240
recHanoi03_T	✓	0.74	34	✓	7.3	190	X	1.8	140	TO	900	880	✓	1.6	52	?	56	320
Score (max. 112)		2			75			-24			64			18			89	
#Correct Results		59			51			56			45			65			59	
#False Positive		0			0			17			0			2			0	
#False Negative		8			0			0			0			6			0	

Table B.2: Verification result, runtime (CPU time in seconds) and memory consumption (in MB) for different tools (correct result: ✓, wrong result: X, timeout: TO, out of memory: OOM, unknown: ?)