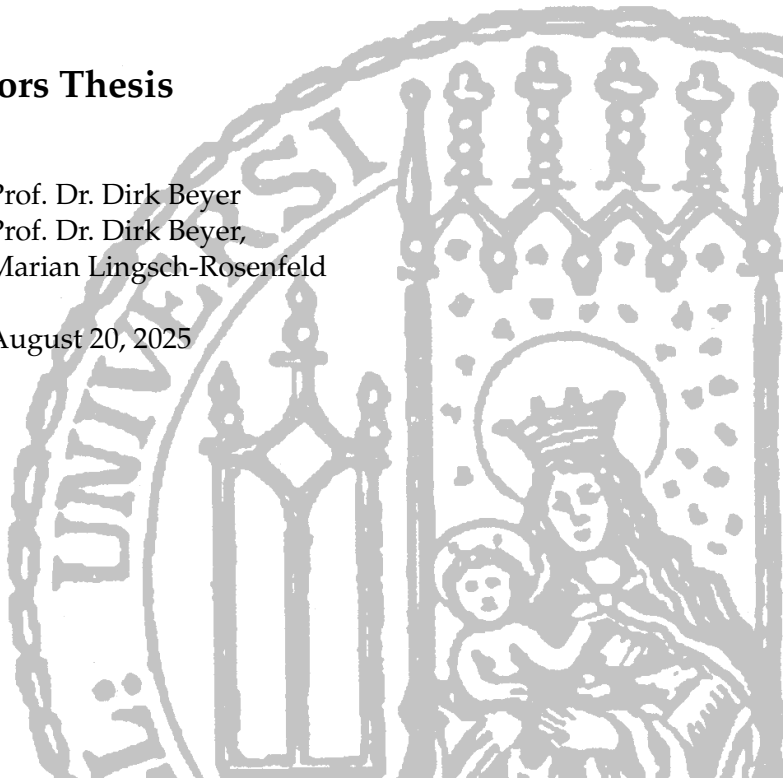
INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

Augmenting Predicate Analysis in CPAchecker using Lemmata

Rabea Lühmann



Bachelors Thesis

Supervisor Prof. Dr. Dirk Beyer
Advisor Prof. Dr. Dirk Beyer,
Marian Lingsch-Rosenfeld

Submission Date August 20, 2025

Statement of Originality

English:

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments.

Deutsch:

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, August 20, 2025

Rabea Lühmann

Rabea Lühmann

Abstract

Software verification has the goal to minimize the risk of software failures with negative consequences. The classical approach to this is deductive verification, which requires considerable involvement from developers. Automatic verification tools seek to automate this process, but there is still a gap between the capabilities of deductive and automatic software verification.

CPACHECKER is an automatic verification framework that combines various formal verification approaches through the concept of *Configurable Program Analysis* (CPA). The *Predicate CPA* for predicate abstraction in CPACHECKER solves a verification task through the construction of an *Abstract Reachability Graph* (ARG).
Functions that operate on cumulative properties over arrays could previously not be verified by CPACHECKER when using predicate abstraction. We propose an extension to the Predicate CPA that uses manually generated lemmas to successfully solve the abstraction step for such verification tasks.

The lemmas are provided to CPACHECKER by the user through a witness file. During an abstraction step all relevant lemmas are added to the first-order formula that is then solved by the SMT solver. We evaluate the performance of our implementation using the benchmarking tool BENCHEXEC. With our prototype we are able to compute the Boolean predicate abstraction for programs that operate on the minimum, maximum and sum of arrays. We show that without a lemma witness the same programs cannot successfully be verified with predicate abstraction in CPACHECKER.
We analyze possible causes for the current limitations of our prototype. For future work we propose the integration of a dedicated ACSL-Parser to improve current limitations of the parsing of a lemma witness. We also suggest to extract our approach into its own *Higher Order Abstraction CPA*.

Zusammenfassung

Software Verifikation hat zum Ziel, das Risiko von Softwarefehlern mit negativen Konsequenzen zu minimieren. Der klassische Ansatz hierfür ist die deduktive Verifizierung, welche erhebliches Mitwirken von Entwicklern erfordert. Automatische Verifizierungswerkzeuge versuchen diesen Prozess zu automatisieren, aber es gibt noch immer eine erhebliche Lücke zwischen den Fähigkeiten von deduktiver und automatischer Softwareverifizierung.

CPACHECKER ist ein automatisches Verifizierungsframework, das verschiedene formale Verifizierungsansätze durch das Konzept der *configurable program analysis* (CPA) kombiniert. Der *Predicate CPA* für predicate abstraction löst eine Verifizierungsaufgabe durch die Konstruktion eines *abstract reachability graph* (ARG). Funktionen, die auf kumulativen Eigenschaften von Arrays operieren, konnten bisher von CPACHECKER nicht durch prediate abstraction verifiziert werden. Wir schlagen einen Erweiterung des Predicate CPA vor, die manuell erstellte Lemmata nutzen kann, um den Abstraktionschritt für solche Programme erfolgreich zu lösen.

Die Lemmata werden CPACHECKER vom Nutzer durch eine witness Datei zur Verfügung gestellt. Während eines Abstraktionsschrittes werden alle relevanten Lemmata der Formal erster Ordnung hinzugefügt, welche dann durch den SMT-Solver gelöst wird. Mit unserem Prototypen sind wir in der Lage, die Boolean predicate abstraction für Programme zu berechnen, die auf dem Minimum, Maximum oder der Summe von Arrays operieren. Wir zeigen, dass dieselben Programme ohne einen lemma witness nicht erfolgreich durch predicate abstraction in CPACHECKER verifiziert werden können.
Wir analysieren mögliche Ursachen für die aktuellen Limitierungen unseres Prototypen. Für zukünftige Arbeiten schlagen wir die Integration eines dedizierten ACSL-Parsers vor, um derzeitige Limitierungen des Parsen eines lemma witness zu verbessern. Außerdem schlagen wir vor, unseren Ansatz in einen eigenen *Higher Order Abstraction CPA* zu extrahieren.

Contents

List of Figures

CHAPTER 1

Introduction

Program code is written by humans and is therefore subject to human error. If such an error affects a critical infrastructure system, the consequences can be disastrous. A prominent example for a critical software failure is the *Altona Railway Software Glitch* [18] of 1995 that stopped all train traffic at one major German railway hub for a duration of four days. On Sunday March 12, 1995 the existing switch tower at Hamburg-Altona was replaced by a fully computerized system manufactured by Siemens that was incompatible with the preexisting one. Immediately upon startup the central computer failed. Because there was no manual mode the operators had no choice but to turn off the whole system and close the station. At the time, a combined 130.000 daily passengers moved through the station to use national and international long distance connections, regional trains and local rapid transit. The bug at fault for the failure was not detected until Tuesday evening and could only be fixed by Wednesday morning. Eventually, its cause was determined to be a programming error in a routine that was supposed to handle stack overflows. Limited traffic at the station picked up again on Wednesday afternoon but it took several days until the station was operating at full capacity again. A minor programming error rendered one of the biggest transit hubs of the country useless for several days and subsequently negatively impacted public opinion of the computerization of infrastructure.

For many software projects, similar to the infrastructure system deployed in Hamburg-Altona, the C programming language [52] is to this day a popular choice. C is a low-level language that allows for fast and efficient access of hardware resources, which makes it a good choice for the software components of Cyber-Physical-Systems [17]. At the same time, the low-level nature of C means that it offers little protection from programming errors.

If we want to minimize the risk of catastrophic software failures, we must ensure that a given computer program is free of errors. Two different approaches to achieve this are software testing and software verification. While software testing can help us find bugs in our software it cannot prove their absence. Software verification however can prove that a certain program fulfills its specification.

A classical approach to software verification is through deductive verification [37], a process that is based on logical inference about verification contracts. Deductive verification is very powerful to prove the formal correctness of properties but it is also a very involved process that requires considerable effort from developers. Even though tools such as FRAMA-C [23] or DAFNY [46] support deductive verification it still requires a lot of time and effort from software developers. For this reason, automatic verification tools seek to automate the process of software verification to reduce the needed effort and make it feasible for usage a greater number of projects. Ultimately, this improves the quality of deployed software.

One such tool for the automatic verification of C code is CPACHECKER [11]. CPACHECKER is easily configurable and expandable and makes it possible to use the same verification framework for a variety of formal approaches [8], like bounded model checking, k-induction or predicate abstraction. The concept at the core of CPACHECKER is that of *Configurable Program Analysis* (CPA) [9]. A CPA defines the abstract domain and the operators that the CPA algorithm calls to solve a verification task. As a result a new CPA can be created by providing a concrete implementation for these interfaces. Similarly, already existing CPAs can be combined into composite CPAs that provide new functionality. For example the CPA for Predicate Abstraction [8, 56] that we will be referring to in the following chapters is a composite of one CPA that tracks the program counter and one CPA performs the predicate abstraction [34].
The current state of predicate abstraction in CPACHECKER can verify some properties of iterative loops. However, loops that deal with cumulative properties of arrays, can not yet successfully be verified by CPACHECKER when using predicate abstraction, because we cannot sufficiently express those properties as abstraction predicates. In the deductive verification tool FRAMA-C we can verify these same array properties through contract based verification. This is a gap in the capabilities of deductive and automatic verification tools. In this thesis we propose an extension to predicate abstraction in CPACHECKER with which we can compute the abstraction step for functions that operate on arrays. Our prototype adds user generated lemmas that provide the necessary definitions to the abstraction predicates, so that the SMT solver has all the context to compute a new abstraction.

Chapter 2 presents a selection of interesting related work and Chapter 3 introduces the theoretical background for predicate abstraction in CPACHECKER. In Chapter 4 we describe our implementation which we evaluate in Chapter 5. In Chapter 6 we discuss the current limitations of our prototype. Afterwards, we give some suggestions for future work in 7. Finally, we summarize the content of this thesis in Chapter 8.

CHAPTER 2

Related Work

In this chapter, we present some relevant work that relates to the content of this thesis. First, we discuss formal methods, which are the basis for software verification. Then, we give an overview of deductive verification and relevant tools. Finally, we consider automatic verification, predicate abstraction and decidability of arrays.

Formal methods are the basis for modern day software verification. Floyd and Hoare [33, 39] introduced a formal logic to verify the correctness of a program S through a Hoare triple $\{P\}S\{Q\}$ with a precondition P and a postcondition Q. In *Guarded commands, non-determinacy and formal derivation of programs* [27] Dijkstra introduced predicate transformer semantics as a method to solve a Hoare triple. If the execution order of the program is forwards the predicate transformer is the strongest postcondition [30], if the execution order is backwards it is the weakest precondition [28]. Predicate abstraction in CPACHECKER makes use of the strongest postcondition to verify a program.

Hoare logic and predicate transformer semantics are the basis for the field of *deductive software verification* for which the paper *Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools* [37] gives a summary from its beginnings in the 1960's to modern day industrial verification tools. The first deductive verification tool was ANNA [47] for the language Ada, then came EIFFEL [50] as the first contract based verification tool. Some tools that are still relevant today are the *Java Modeling Language* JML [41, 45], KEY [1] and the proof assistant ISABELLE [57]. DAFNY [46] is a programming language that requires verification alongside the writing of the program code. The DAFNY source code can then be compiled into popular programming languages like Java, JavaScript, C#, Python and Go.

Another contract based tool is for the deductive verification of C code is FRAMA-C [22, 23]. The paper *An Exercise in Mind Reading: Automatic Contract Inference for Frama-C* [3] presents the FRAMA-C plugin SAIDA that can automatically infer ACSL-Annotations [7, 22] for all helper functions of a given C program. SAIDA takes as input a C program with an already annotated main function as the entry point and a specification as input and transfers it into constrained horn clauses (CHC) [26]. The plugin is intended to automate the very difficult and time intensive process of manually constructing contracts for every function.
Program specifications are often written in high-level languages like first order logic to express the intended program behavior correctly and precisely. However, the solvers used as the back-end of a verification tool often do not support the complex constructs of the specification language. The paper *Automatic Program Instrumentation for Automatic Verification* [2] presents MONOCERA, a framework to automatically translate rich program specification into a low-lever verification language consisting of program independent instrumentation operations. The proposed language contains instrumentation operators to represent universal and existential quantifiers and can handle arrays and array aggregation. Because the implementation is based in constrained horn clauses [26], it is available for a variety of CHC-based verification tools like JAYHORN [42], KORN [31], RUSTHORN [48], SEAHORN [35], and TRICERA [32].

Automatic verification tools like CPACHECKER [8, 9, 11], SLAM [5, 6] or BLAST [10] aim to reduce the overhead of deductive verification through automation. There are several approaches to automatic software verification that are implemented in CPACHECKER and related tools. *Bounded model checking* can be used by CBMC [20], ESBMC [21], LLBMC [53] and SMACK [51]. *Unbounded model checking* can be done via *k-Induction* [55] in tools like CBMC [20], ESBMC [21], PKIND [43] and 2LS [16]. A different approach to automatic verification from model checking is through an over approximation of the state space, either by *predicate abstraction* [34] or the IMPACT algorithm [49]. Predicate abstraction is usually combined with *counter example guided abstraction refinement* (CEGAR) [19] and *lazy abstraction* [38]. CPACHECKER combines all of the afformentioned approaches to automatic software verification within an unified framework. *Towards Practical Predicate Analysis* [56] introduces predicate abstraction for CPACHECKER by implementing the *Predicate CPA*, which we build upon in this thesis. The following paragraphs give an overview over a selection of papers that also use predicate abstraction.

The paper *Automatic Predicate Abstraction of C Programs* [4] presents the tool C2BP [5] as part of the SLAM toolkit. The implementation is the first algorithm that can automatically construct a predicate abstraction for device drivers as well as array and pointer manipulating programs written in C.

In *Experience with Predicate Abstraction* [25], the authors present $Mur\phi^{--}$ a simplification of the $Mur\phi$ language. Through their implementation, they are able to verify the FLASH multiprocessor cache coherence protocol and Dijkstra's on-the-fly garbage collection algorithm [29]. Furthermore, the implementation offers limited support for quantifiers.

The majority of efforts to develop SMT-solving to support arrays has been focusing on safety properties. *Regular Abstractions for Array Systems* [40] combines string rewriting systems [15] with a new predicate abstraction to verify safety and liveness properties over arrays. The paper *Constructing Quantified Invariants via Predicate Abstraction* [44] presents a predicate abstraction that constructs a formula of universally quantified variables. This abstraction can be used to describe large memories, buffers and arrays.

To express the properties of arrays in first-order logic, we need quantifiers which makes the full first-order theories of arrays undecidable [16] in SMT-solving. *Array Folds Logic* [24] defines a new logic called AFL that can express properties over arrays that had not previously been covered by decidable fragments of array theories and cannot be expressed in first-order logic. Other papers that deal with the decidability of array properties are *Verification Decidability of Presburger Array Programs* [54], *What's Decidable About Arrays?* [16] and *What Else Is Decidable about Integer Arrays?* [36].

CHAPTER 3

Background

The main components of CPACHECKER are a *Configurable Program Analysis* (CPA) [8] and the *CPA Algorithm*. To solve a verification task the CPA Algorithm uses operators whose concrete implementation is provided by a possibly composite CPA. The CPA for predicate abstraction is a composite CPA that constructs an *Abstract Reachability Graph* (ARG) [8] by mapping abstract states over predicates to concrete locations in the programs *Control Flow Automaton* (CFA) [8]. Our implementation makes changes to one of the component CPAs for predicate abstraction, the *Predicate CPA*.
In this chapter we introduce the operators of the *Predicate CPA* and explain their roles in constructing the ARG. In Chapter 4 we present the changes that we make to the *Predicate CPA* with which we build upon the theoretical background given in this chapter. During the evaluation of our approach in Chapter 5 we discuss the limitations of our prototype. We analyze possible causes for these limitations in Chapter 6. To understand the limitations and how they can be improved it is important to know the theoretical background we introduce in this chapter.

In Sect. 3.1 we introduce the *Control Flow Automaton* (CFA) which represents concrete locations in the source code in CPACHECKER. Sect. 3.2 gives a short introduction for the *CPA Algorithm*. We explain the concept of *Configurable Program Analysis* in Sect. 3.3 and define the operators for the *Predicate CPA* in Sect. 3.3.1. In Sect. 3.4 we give an example for the usage of the operators from Sect. 3.3.1 by constructing the ARG for an example program.

3.1 Control Flow Automata

The representation of the source program in CPACHECKER is a *Control Flow Automaton* (CFA) [8]. A CFA $A = (L, l_{init}, G)$ is a directed graph with the nodes representing program locations from the set L and its edges from the set G representing operations between two program locations. The initial location $l_{init} \in L$ is the entry point of the program or function. An operation along a CFA edge $G \subseteq (L \times Ops \times L)$ can either be an arithmetic assignment over integers or an assume operation over a predicate of variables from the program.

The CFA in Fig. 3.1represents the function `maxArray(int* a, int l, int i)` from the C program `maxArray.c` in Listing 1. The function takes as input an array of unsigned integers `a`, the length of the array `l` and an index `i` between zero and the length of the array. It computes the maximum value of the array by iterating over all elements in `a` in ascending order and comparing the current element to the previous maximum `m`. The program reaches an error state, if the maximum after completion of the loop is smaller than the array element at index `i`. Otherwise it returns the maximum `m` of the array.
We want to prove with predicate abstraction in CPACHECKER, that the error state in line `16` can never be reached and as a consequence the correctness of the computed maximum. To achieve this goal we assign the external predicates `1 <= j <= l` and `m = Max(a,j)` to the head of the while loop in line `9`. The inductive predicate `Max(A,I)` is defined as `Max(A,0) = A[0]` and `Max(A,I) = A[I] > Max(A,I-1) ? A[I] : Max(A,I-1)`.

Listing 1 C program that finds the maximum from an array

```
1  int maxArray(int* a,unsigned int l, unsigned int i) {
2    int j = 0;
3    unsigned int m = a[0];
4    while(j < l) {
5      if(a[j] > m) {
6        m = a[j];
7      }
8      j = j+1;
9    }
10   if(m < a[i]) {
11     reach_error();
12     return -1;
13   }
14   return m;
15 }
```

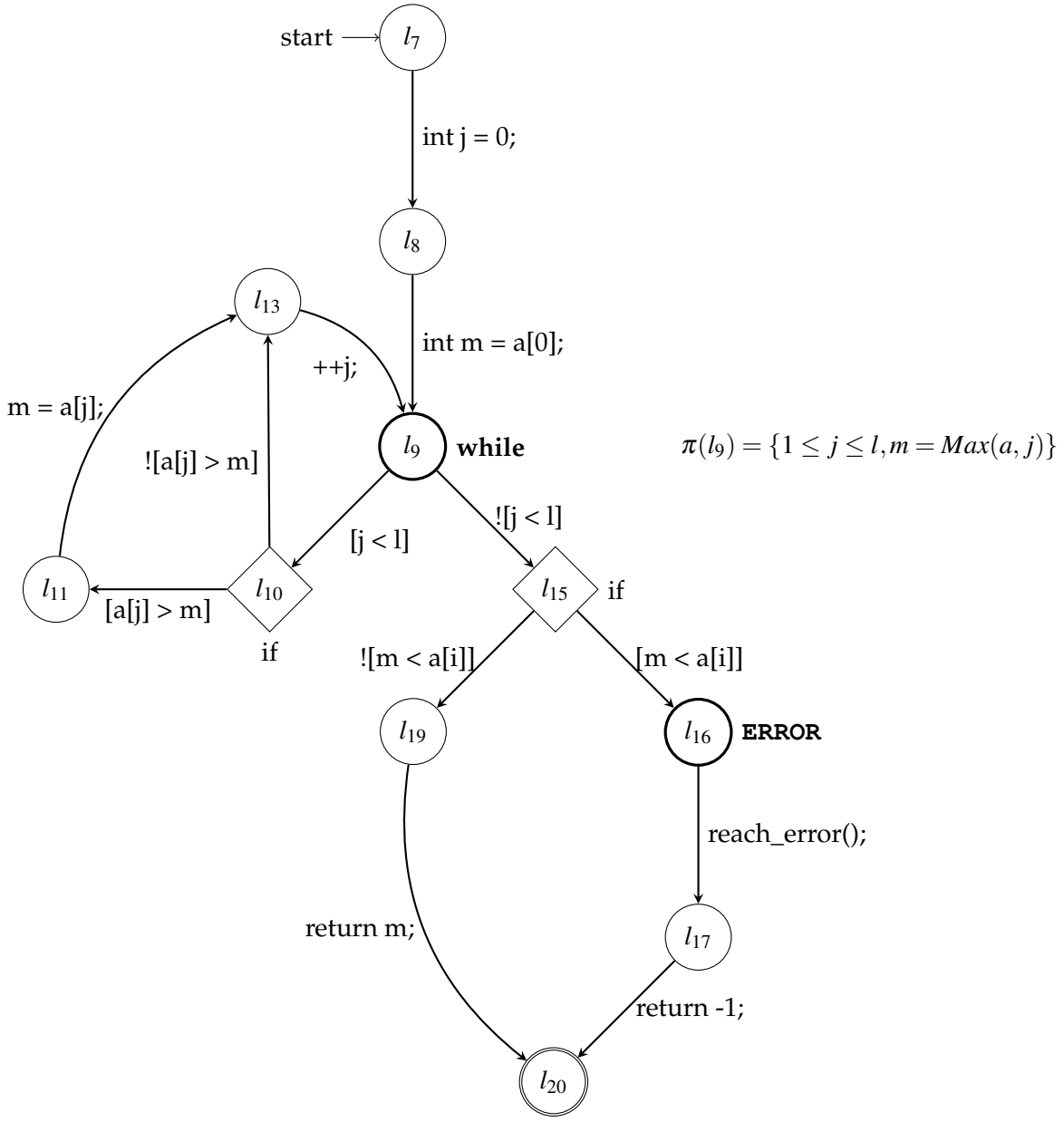


Figure 3.1: CFA representing the function from Listing 1

3.2 CPA Algorithm

The CPA algorithm [8] shown in Algorithm 1 uses a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, merge, stop, prec)$ and an initial abstract state e_{init} with precision π_{init} as input. The algorithm returns a set of all reachable abstract states. The implementation for the merge operator *merge*, the stop operator *stop*, the precision adjustment operator *prec* and the *transfer relation* $\rightsquigarrow$ is provided by the CPA $\mathbb{D}$. We discuss the definition of the operators for predicate abstraction in Sect. 3.3.1.

While the set waitlist is not empty, the algorithm takes an abstract state e with precision π from the waitlist and calls the precision adjustment operator *prec* on it. At an abstraction location the *prec* operator computes a new abstraction state with an updated precision. At other locations it leaves the abstract state e unchanged.
Then the algorithm calls the transfer relation to calculate all abstract successors of the precision adjusted abstract state $\widehat{e}$. Each abstract successor is merged with every abstract state within the reached set. If the merged state differs from the one in reached, the new state replaces the old one in both the *waitlist* and *reached* sets.
The stop operator decides whether the abstract successor e' of $\widehat{e}$ is implied by another abstract state from the reached set. If this is the case, e' does not need to be explored further. Otherwise it gets added to both the reached set and the waitlist.
When all abstract states from the waitlist have been evaluated, the algorithm returns the set of reachable abstract states. If the final reached set does not contain an abstract error state then the error location of the CFA can not be reached while executing the program.

Algorithm 1 CPA+$(\mathbb{D}, e_{init}, \pi_{init})$, taken from [8]

```
waitlist := {(e_init, π_init)}
reached := {(e_init, π_init)}
while waitlist ≠ ∅ do
  pop (e, π) from waitlist
  (ê, π̂) := prec(e,π,reached)
  for all e' with ê ⇝ (e', π̂) do
    for all (e'', π'') ∈ reached do
      e_new := merge(e', e'', π̂)
      if e_new ≠ e'' then
        waitlist := (waitlist ∪{(e_new, π̂)}) \ {(e'', π'')})
        reached := (reached ∪{(e_new, π̂)}) \ {(e'', π'')})
      if not stop(e', {e|(e,·) ∈ reached},π̂) then
        waitlist := waitlist ∪{(e', π̂)}
        reached := reached ∪{(e', π̂)}
return {e|(e,·) ∈ reached}
```

3.3 CPAchecker's Predicate Abstraction

The CPA for predicate abstraction is a CPA that is composed of the *Location CPA* $\mathbb{L}$ that tracks the program counter and the *Predicate CPA* $\mathbb{P}$ that operates on abstract states [8] of predicates over the program domain. Combining the Location CPA $\mathbb{L}$ with the Predicate CPA $\mathbb{P}$ produces an *Abstract Reachability Graph* (ARG) [8] that stores the predecessor-successor relationship between abstract states and maps those abstract states to concrete program locations. This way we can determine if and on which program path a program location is reachable at runtime.
The definition for the operators of the Predicate CPA $\mathbb{P}$ is given in Sect. 3.3.1 and the resulting ARG for the function `maxArray` Fig. 3.1 is explained in Fig. 3.2.

3.3.1 The Predicate CPA

The Predicate CPA $\mathbb{P} = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, merge_{\mathbb{P}}, stop_{\mathbb{P}}, prec_{\mathbb{P}})$ [8] defines the transfer relation and the operators *merge*, *stop*, *prec* of the CPA algorithm for an abstract state $e = (\psi, l^{\Psi}, \phi)$. An abstract state is a representation of all concrete states that fulfill the conjunction of the abstraction formula ψ and the path formula ϕ which are first order formulas over variables from the program. The abstraction formula is computed by the predicate precision adjustment operator *prec* only at certain interesting program locations while the path formula is updated by every call of the predicate transfer relation $\rightsquigarrow_{\mathbb{P}}$. The abstraction location l^{Ψ} stores the location at which the current ψ was computed. If an abstract state is of the form $e = (\psi, l^{\Psi}, true)$ it is called an abstraction state, otherwise it is called an intermediate state.

Definition 1 (Predicate Transfer Relation)**.** The *transfer relation* $\rightsquigarrow_{\mathbb{P}}$ [8] computes the abstract successor state $e' = (\psi, l^{\Psi}, \phi')$ for an abstract state $e = (\psi, l^{\Psi}, \phi)$ and a CFA edge (l_i, op_i, l_j) by applying the strongest-postcondition operator $SP_{op_i}(\phi)$ [30] for the operation op_i to the path formula ϕ of e.

$$(\psi, l^{\Psi}, \phi) \rightsquigarrow_{\mathbb{P}} ((\psi, l^{\Psi}, \phi'), \pi) \text{ for a CFA edge } (l_i, op_i, l_j)$$

The operation op_i along the edge can either be an arithmetic assignment $x := e$ of a value to a program variable or an assume statement over a predicate p over program variables:

$$SP_{x:=e}(\phi) = \exists x_0 : \phi_{x \to x_0} \wedge (x = e_{x \to x_0})$$
$$SP_{[p]}(\phi) = \phi \wedge p$$

Definition 2 (Merge Operator)**.** If two abstract states have the same abstraction formula ψ and the same abstraction location l^{Ψ} they can be merged into one abstract state with the new path formula ϕ being the disjunction of the two original path formulas.

$$merge_{\mathbb{P}}((\psi_1, l_1^{\Psi}, \phi_1), (\psi_2, l_2^{\Psi}, \phi_2), \pi) \tag{3.1}$$
$$= \begin{cases} (\psi_2, l_2^{\Psi}, \phi_1 \vee \phi_2) & \text{if}(\psi_1 = \psi_2) \wedge (l_1^{\Psi} = l_2^{\Psi}) \\ (\psi_2, l_2^{\Psi}, \phi_2) & \text{otherwise} \end{cases} \tag{3.2}$$

Definition 3 (Stop Operator)**.** The stop operator $stop_{\mathbb{P}}$ [8] determines whether an abstract state $e = (\psi, l^{\Psi}, true)$ is implied by an abstract state $e' = (\psi', l'^{\Psi}, true)$ from the reached set. If the abstract state e is already covered by the reached state e' it does not need to be explored further.

$$stop_{\mathbb{P}}((\psi, l^{\Psi}, \phi), R, \pi) \tag{3.3}$$

$$= \begin{cases} \exists (\psi', l^{\Psi'}, \phi') \in R : \phi' = true \wedge (\psi, l^{\Psi}, \phi) \sqsubseteq_{\mathbb{P}} (\psi', l^{\Psi'}, \phi') & \text{if } \phi = true \\ false & \text{otherwise} \end{cases} \tag{3.4}$$

With the partial order $\sqsubseteq_{\mathbb{P}}$ defined as: (3.5)

$$(\psi_1, l_1^{\Psi}, \phi_1) \sqsubseteq_{\mathbb{P}} (\psi_2, l_2^{\Psi}, \phi_2) = ((\psi_1 \wedge \phi_1) \Rightarrow (\psi_2 \wedge \phi_2)) \tag{3.6}$$

Definition 4 (Precision Adjustment Operator)**.** The predicate precision adjustment operator $prec_{\mathbb{P}}$ [8] performs a Boolean predicate abstraction at certain program locations of interest. In our case for the function `maxArray` these *abstraction locations* are the head of the while loop in line `9` and the call of the error function `reach_error()` in line `16`.
At an abstraction location the predicate precision adjustment takes an abstract state e, a *predicate precision* π [8] and the reached set R and returns an abstraction state and an updated precision. The new abstraction formula ψ is the result of the Boolean predicate abstraction $(\phi)_{\mathbb{B}}^{\pi(l)}$ and the new abstraction location is the current location.

$$prec_{\mathbb{P}}((\psi, l^{\Psi}, \phi), \pi, R) \tag{3.7}$$

$$= \begin{cases} (((\psi \wedge \phi)_{\mathbb{B}}^{\pi(l)}, l, true), \pi) & \text{if } blk((\psi, l^{\Psi}, \phi), l) \\ ((\psi, l^{\Psi}, \phi), \pi) & \text{otherwise} \end{cases} \tag{3.8}$$

The block adjustment operator *blk* [8] decides, whether an abstraction step should be performed for an abstract state at the current program location. Common choices are blk^{lf} and blk^{l}:

$$blk^{lf} = \begin{cases} true & \text{at loop heads, function calls and return statements, } l_{err} \\ false & \text{otherwise} \end{cases} \tag{3.9}$$

$$blk^{l} = \begin{cases} true & \text{at loop heads and } l_{err} \\ false & \text{otherwise} \end{cases} \tag{3.10}$$

(3.11)

If *blk* returns *true* the Boolean predicate abstraction $(\pi \wedge \phi)_{\mathbb{B}}^{\pi(l)}$ is computed by an SMT solver by solving the equation:

$$(\psi)_{\mathbb{B}}^{\rho} = \psi \wedge \bigwedge_{p_i \in \rho} (v_{p_i} \Leftrightarrow p_i) \tag{3.12}$$

For the example program `maxArray.c` with blk^{l} the precision adjustment is calculated at the head of the while loop in line `9` and the error location in line `16`. An example for this is given in Sect. 3.4.

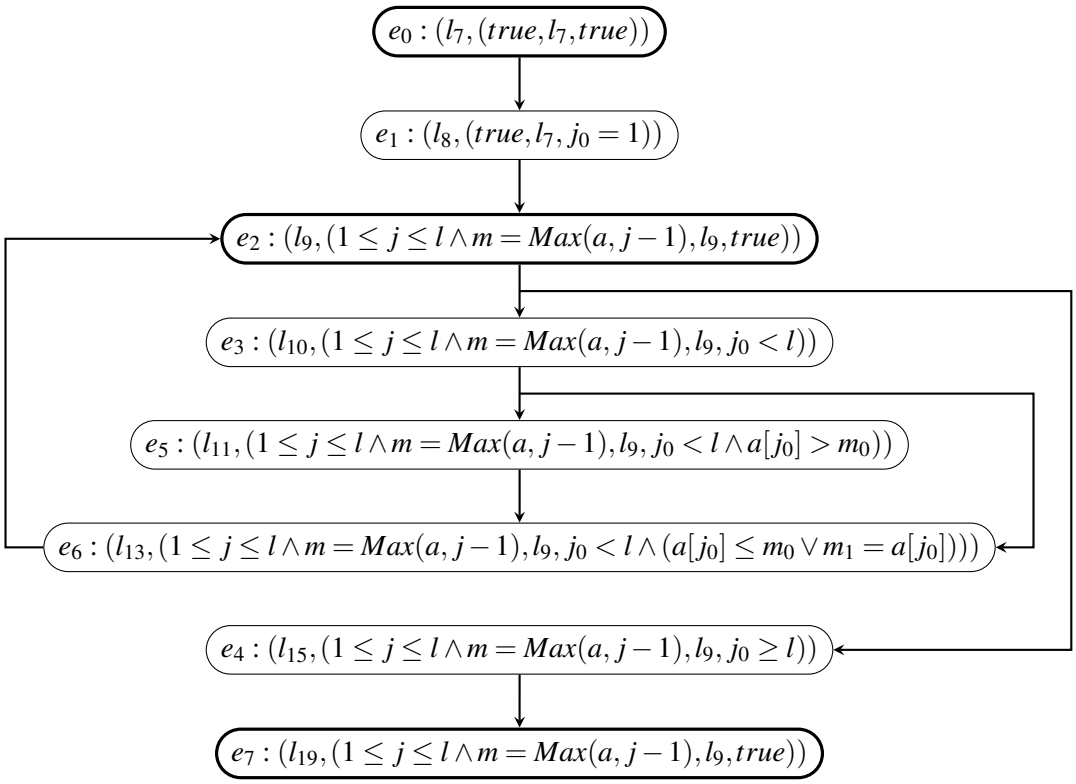


Figure 3.2: ARG for the CFA locations from Fig. 3.1

3.4 Example

We want to construct the abstract reachability graph Fig. 3.2 for the function `maxArray` Fig. 3.1. At the start of our analysis the reached set R and the waitlist both contain only the initial ARG state $e_0 : (l_7, (true, l_7, true))$. After a couple iterations of the CPA algorithm we arrive at the configuration $reached = \{e_0, e_1, e_2, e_3, e_4\}$ and $waitlist = \{e_3, e_4\}$ where the intermediate state $e_3 : (l_{10}, (1 \leq j \leq l \wedge m = Max(a, j-1), l_9, j_0 < l))$ should be evaluated next. If we look at the CFA Fig. 3.1 we can see that line `10` is an if-statement with the outgoing edges $(l_{10}, a[j] > m, l_{11})$ and $(l_{10}, \neg(a[j] > m), l_{13})$. Applying the transfer relation for each of those CFA edges to the ARG state $e_3 : (l_{10}, (1 \leq j \leq l \wedge m = Max(a, j-1), l_9, j_0 < l))$ produces the following successor states:

$$e_5 = (l_{11}, (1 \leq j \leq l \wedge m = Max(a, j-1), l_9, j_0 < l \wedge a[j_0] > m)) \tag{3.13}$$
$$e'_6 = (l_{13}, (1 \leq j \leq l \wedge m = Max(a, j-1), l_9, j_0 < l \wedge a[j_0] \leq m)) \tag{3.14}$$

Applying the transfer relation again to e_5 produces the successor:

$$e'_5 = (l_{13}, (1 \leq j \leq l \wedge m = Max(a, j-1), l_9, j_0 < l \wedge a[j_0] > m \wedge m_1 = a[j_0])) \tag{3.15}$$
$$\tag{3.16}$$

The states e'_5 and e'_6 share the same program counter, abstraction formula and abstraction location which means they can be merged into the ARG state e_6.

$$merge_{ARG}(e'_5, e'_6,) = (l_{13}, (1 \leq j \leq l \wedge m = Max(a, j-1), l_9, \phi'_5 \vee \phi'_6)) \tag{3.17}$$

$$= (l_{13}, (1 \leq j \leq l \wedge m = Max(a, j-1), l_9, j_0 < l \wedge (a[j_0] \leq m_0 \vee m_1 = a[j_0]))) \tag{3.18}$$

The successor state e''_6 of e_6 along the CFA edge $(l_{13}, j = j+1, l_9)$ is:

$$e''_6 = (l_9, (1 \leq j \leq l \wedge m = Max(a, j-1), l_9, j_0 < l \wedge (a[j_0] \leq m_0 \vee m_1 = a[j_0]) \wedge j_1 = j_0 + 1)) \tag{3.19}$$

Line 9 as the head of the while loop is an abstraction location. Therefore the precision adjustment operator will perform a Boolean abstraction for the ARG state e''_6 and the precision $\pi(l_9) = \{1 \leq j \leq l, m = Max(a, j-1)\}$. For this we need the abstraction formula ψ to be instantiated with the oldest variables from the path formula ϕ. For each predicate from π we introduce a fresh variable v_{p_i} that is true if and only if the instantiated predicate $p_i \in \pi$ is true. Each predicate p_i needs to be instantiated with the newest variables from ϕ.

$$(\psi \wedge \phi)_{\mathbb{B}}^{\pi(l_9)} = 1 \leq j_0 \leq l \wedge m_0 = Max(a, j_0 - 1) \tag{3.20}$$

$$\wedge\, j_0 < l \wedge (a[j_0] \leq m_0 \vee m_1 = a[j_0]) \wedge j_1 = j_0 + 1 \tag{3.21}$$

$$\wedge\, (v_{p_1} \Leftrightarrow 1 \leq j_1 \leq l \wedge v_{p_2} \Leftrightarrow m_1 = Max(a, j_1 - 1)) \tag{3.22}$$

The first predicate v_{p_1} evaluates to true.

$$1 \leq j_0 < l \wedge j_1 = j_0 + 1 \Rightarrow (v_{p_1} \Leftrightarrow 1 \leq j_1 \leq l) = true \tag{3.23}$$

The second predicate v_{p_2} also holds.

$$m_1 = a[j_1 - 1] \wedge a(j_1 - 1) > Max((j_1 - 1) - 1) \vee m_1 = Max((j_1) - 1) \tag{3.24}$$

$$\wedge\, j_1 = j_0 + 1 \wedge m_0 = Max(a, j_0 - 1) \wedge (a[j_0] \leq m_0 \vee m_1 = a[j_0]) \tag{3.25}$$

$$\Rightarrow m_1 = a[j_0] \wedge a[j_0] > m_0 \vee m_1 = m_0 \wedge (a[j_0] \leq m_0 \vee m_1 = a[j_0]) \tag{3.26}$$

$$\Rightarrow (v_{p_2} \Leftrightarrow m_1 = a[j_0] \wedge a[j_0] > m_0 \vee m_1 = m_0) = true \tag{3.27}$$

The resulting ARG state is $e'_2 = (l_9, (1 \leq j \leq l \wedge m = Max(a, j-1), l_9, true)$. This state is already covered by the state e_2 from the reached set, so there is no need for e'_2 to be unrolled further. The *stop* operator returns *true* and the state e'_2 does not get added to the reached set and the waitlist. The analysis continues with $reached = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6\}$ and $waitlist = \{e_4\}$. So far we have determined that the precision $\pi(l_9)$ holds for the while-loop that calculates the maximum of an array. Now we are looking at an abstract at line 15 after the while-loop. We want to determine if the next abstraction location, the `reach_error()` call in line 16 is reachable at runtime.

CHAPTER 4

Approach

In Chapter 3 we have discussed the *CPA Algorithm* and its operators $merge_{\mathbb{P}}$, $stop_{\mathbb{P}}$ and $prec_{\mathbb{P}}$ and the transfer relation $\rightsquigarrow_{\mathbb{P}}$ as they are defined for the *Predicate CPA* in Sect. 3.3.1. As an exemplary use for these operators, in Sect. 3.4 we have constructed the *Abstract Reachability Graph* (ARG) for the function `max-array()` from Listing 1. At an abstraction location, the precision adjustment operator $prec_{\mathbb{P}}$ computes the *Boolean predicate abstraction* $(\psi)^{\rho}_{\mathbb{B}}$ by solving Eq. (4.1). In this equation the *primary formula* ψ is the conjunction of the *abstraction formula* as an instantiated formula and the path formula which is always instantiated. For every relevant predicate form the predicate precision we initialize a fresh *symbolic variable* v_{p_i}. For each predicate we then build an equivalence between the symbolic variable v_{p_i} and its *symbolic atom* p_i, which is the definition of the predicate. For our example function `max-array()` the symbolic atom from our predicate witness Listing 2 is `m == _L_MAX(a,j)`.

$$(\psi)^{\rho}_{\mathbb{B}} = \psi \wedge \bigwedge_{p_i \in \rho} (v_{p_i} \Leftrightarrow p_i) \tag{4.1}$$

To solve this step for our example function that calculates the maximum of an array we need to expand Eq. (4.1) with a set of lemmas L that provide the definition of the inductive predicate `_L_MAX` that is called by the predicate. To make sense of the lemmas that might appear in the abstraction formula we conjunct the set of lemmas L with the primary formula ψ. For each predicate $p_i \in \rho$ we need the corresponding lemmas $l_{ij} \in L$. All variables that appear in the predicate lemmas need to be instantiated with the same values as the corresponding variables from the predicates which gives us Eq. (4.2):

$$(\psi)^{\rho}_{\mathbb{B}} = (\psi \wedge \bigwedge_{L}) \wedge \bigwedge_{p_i \in \rho} ((v_{p_i} \Leftrightarrow p_i) \wedge \bigwedge_{l_j \in L} l_{ij}) \tag{4.2}$$

To solve the Boolean predicate abstraction a user provided predicate needs to be supplemented by a set of also user generated lemmas. Providing user generated predicates via a witness file was already possible in the current state of CPACHECKER but providing additional lemmas was not. The changes to CPACHECKER that we propose with this thesis make it possible to use lemmas to successfully compute a Boolean predicate abstraction that can otherwise not be solved via predicate abstraction.

The lemmas need to be provided to CPACHECKER via manually generated witness files as described in Sect. 4.1. The content of these witnesses then needs to be made part of the CPACHECKER class `PredicatePrecision`. The parsing of the lemmas to our `max-array()` function poses some challenges with the current state of CPACHECKER which will be discussed in Sect. 4.2. With the lemmas part of the `PredicatePrecision`, some additions need to be made to the implementation of the `PredicatePrecisionAdjustment` class, so the lemmas can be used to successfully solve an abstraction step. An overview of those changes is presented in Sect. 4.3.

Our approach brings the following additions to the implementation of CPACHECKER:

- Parsing lemmas from a witness file into an `AbstractionLemma`
- Adding a set of `AbstractionLemma` to the `PredicatePrecision`
- At an abstraction location:
  - Selecting the relevant lemmas for the primary formula
  - Appending the selected lemmas to the primary formula
- For a Boolean Abstraction:
  - Selecting the relevant lemmas for the relevant predicates
  - Instantiating the lemmas the same way as the predicates
  - Adding the instantiated lemmas to the solver context

4.1 The Lemma Witness

Users can provide abstraction lemmas to CPACHECKER via witness files in the YAML [1] format. The Listing 2 shows a simple example of a lemma witness in lines `15` to 23. A lemma witness is defined by a list of one or more entries where the `entry_type` is `lemma_set`. In addition to a list of one or more lemmas, a `lemma_set` also contains a list of all function and variable declarations that are appear within its lemmas. This information is encoded by two key-value pairs: The key `declarations` maps to a sequence of all declarations in the form of valid C declarations. The key `content` contains a list of individual lemma entries.
Each lemma from `content` is marked by the keyword `lemma` and has itself two key-value pairs. The field `value` contains the body of the actual lemma in the format of a C expression. A lemma can contain one or more function calls but the parsing of function calls into data structures is not supported by the current state of CPACHECKER. This limitation is discussed in more detail in Sect. 4.2. The proposed implementation provides a workaround to circumvent this limitation and make the parsing of lemmas that contain function calls possible.

Because of how the parsing of the lemma witness currently works, all functions that appear in a lemma entry or a predicate need start with the prefix `_L_` and be surrounded by the pattern `ACSL(<function>)`.

The Table 4.1 gives an overview over the structure of the entries of a lemma witness. Listing 2 demonstrates a simple example of a lemma witness with one lemma entry in the lines `15` to `23`.

Key	Value	Description
`entry_type`	`lemma_set`	The entry_type of a lemma witness
`declarations`	sequence	All relevant declarations for the lemma as valid C declarations
`content`	sequence	A sequence of one or more lemma elements
	content of `content`	
`lemma`	mapping	A basic building block of a lemma witness
`value`	scalar	The actual lemma
`format`	`c_expression`	A lemma is a c_expression

Table 4.1: Structure of the entries of a lemma witness

[1] `https://yaml.org/`

Listing 2 An example of a lemma witness

```yaml
- entry_type: invariant_set
  metadata:
    format_version: "2.0"
    producer:
      name: "Handcrafted"
  content:
    - invariant:
        type: loop_invariant
        location:
          line: 18
          column: 3
          function: maxArray
        value: "m == ACSL(_L_MAX(a,j))"
        format: c_expression
- entry_type: lemma_set
  declarations:
    - "int _L_MAX(int* A, int I)"
    - "int* A"
    - "int I"
  content:
    - lemma:
        value: "ACSL(_L_MAX(A,0)) == A[0]"
        format: c_expression
```

4.2 Parsing of the Lemmas

The user provided lemma entries from the witness must be parsed into an internal data structure so they can be used to solve an abstraction problem. This data structure is the newly added class `AbstractionLemma` which is a part of the preexisting `PredicatePrecision`, as shown in Fig. 4.1. For each `lemma_set` entry from a witness an instance of the class `AbstractionLemma` is created. An `AbstractionLemma` has two fields: The set `formulas` contains the content of all `value` entries from the `lemma_set` as Boolean formulas. The String `identifier` is the unique name of the lemma function that is prefixed by `_L_`. The identifier for the `lemma_set` from our example witness Listing 2 is `_L_MAX`.

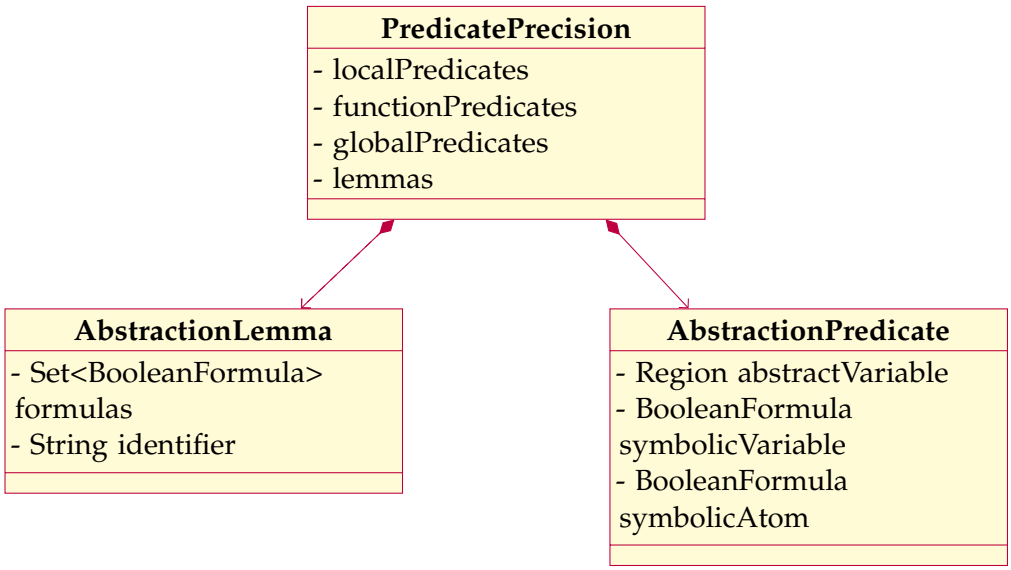


Figure 4.1: The Predicate Precision

In the current implementation of CPACHECKER, to get a `BooleanFormula` from a string of a C expression the content of the string is first parsed into a `CStatement`. The two implementations of `CStatement` that are relevant for the parsing of a lemma are `CExpressionStatement` and `CFunctionCallStatement`, as shown in Fig. 4.2. If the string that we want to parse does not contain any function calls it is parsed into a `CExpressionStatement`. From a `CExpressionStatement` we can derive a `CExpression` `BooleanFormula` by preexisting methods if the expression is a `CBinaryExpression`.

If the string that is to be parsed, contains a C function call, like our lemma `_L_MAX(A,0) == A[0]`, it is parsed into a `CFunctionCallStatement` from which we can get a `CFunctionCallExpression`. But the `CFunctionCallExpression` class does not implement the `CExpression` interface and can therefore not easily be transformed into a `BooleanFormula`. To mitigate this limitation, we introduce the class `ACSLFunctionCall`, which is a `CFunctionCallExpression` that also implements the `CExpression` interface, as shown in Fig. 4.3.

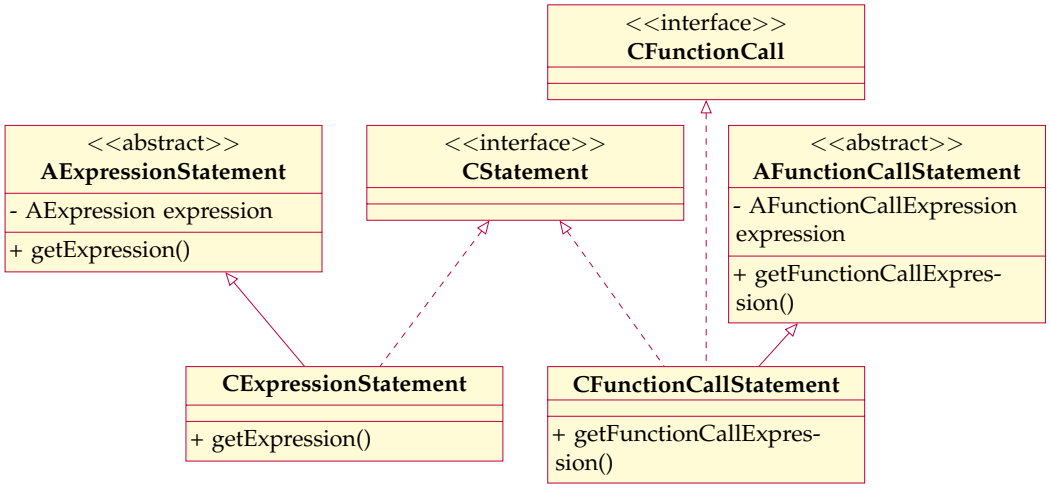


Figure 4.2: Excerpt from the class hierarchy of C statements

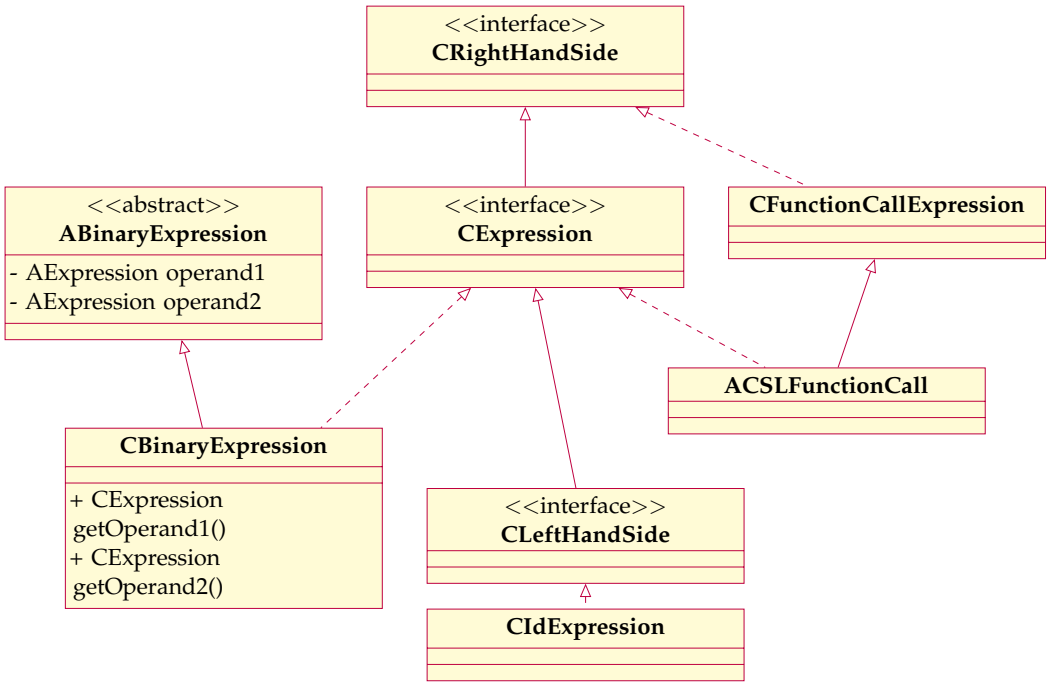


Figure 4.3: Excerpt from the class hierarchy of C expressions

We can derive a `CBinaryExpression` from our lemma string in three general steps: First, all substrings that are function calls are extracted from the lemma string. The function calls in the original string are replaced by placeholder variables that contain no function calls. Then, the function call is parsed into an `ACSLFunctionCall` while the rest of the lemma can be parsed into a `CBinaryExpression` via preexisting methods. Lastly, all occurrences of placeholder variables in the `CBinaryExpression` are replaced by the corresponding `ACSLFunctionCall`. From the `CBinaryExpression` that contains the function calls we can then derive a `BooleanFormula` that can be added to the formulas of an `AbstractionLemma`.

The parsing of a statement that contains function calls is handled by the newly introduced class `ACSLParserUtils`. When parsing a lemma, the method `parseACSLExpression` searches the string for all occurrences of the regular expression `ACSL(*)` that marks the appearance of a function call. For each occurrence of the pattern the function call substring is parsed into an `ACSLFunctionCall`. In the original string the substring `ACSL(*)` is replaced by a variable of the form `lemma_tmp_i`, where the index `i` is increased with every match. The function call is parsed into an `ACSLFunctionCall` and the relationship between thee replacement variable and the `FunctionCall` is stored in a map.
Our example string `ACSL(_L_MAX(A,0)) == A[0]` contains the pattern exactly once. The replaced string `lemma_tmp_1 == A[0]` can be parsed as a regular C Expression that contains no function calls. The substring `_L_MAX(A,0)` is parsed into an `ACSLFunctionCall` by the method `extractFunctionCall()`.

Inserting the function calls back into the `CBinaryExpression` is done by the `ACSLVisitor`. This visitor recursively searches a `CBinaryExpressions` for all occurrences of a placeholder variable of the form `lemma_tmp_i`.
A `CBinaryExpression` has two operands, which are also instances of `CExpression`. For each operand, the visitor checks if it is an instance of `CIdExpression`. If the operand is a `CIdExpression` whose name is a placeholder variable, the visitor replaces this operand with the corresponding `ACSLFunctionCall`. Otherwise, it recursively visits that operand which itself can be a `CBinaryExpression`. After it has traversed the whole expression, the visitor returns a `CExpression` that contains all function calls instead of their placeholders.

From the `CExpression` that is returned by `parseACSLExpression()`, we can derive the `BooleanFormula` for our lemma. The following section discusses how the lemmas are used to compute a new abstraction at an abstraction location.

4.3 Computing a new Abstraction

At an abstraction location, the CPA algorithm calls the predicate precision adjustment operator *prec*. This operator computes a new abstraction state and an updated precision from an abstract state, a reached set and a predicate precision. The theoretical background of precision adjustment is introduced in Definition 4.

The implementation for the *prec* operator in CPACHECKER is provided by the method `prec()` from the class `PredicatePrecisionAdjustment`, which decides if a new abstraction should be computed. The actual computation of a new abstraction is implemented by the class `PredicateAbstractionManager`. The relevant methods for the computation of a Boolean abstraction from this class are: `buildAbstraction()`, `computeAbstraction()` and `computeBooleanAbstraction()`.
In the following section, we will see how these methods have been modified to make use of the abstraction lemmas that are now part of the predicate precision.

The method `buildAbstraction()` instantiates the abstraction formula and conjuncts it with the path formula into a *primary formula*. An abstraction formula can contain predicates which are dependent on abstraction lemmas. For this reason, we now need to select all the relevant lemmas for the abstraction formula and append them to the primary formula. This step is performed by the `LemmaSelectionVisitor`, which is explained in Sect. 4.4. Next, `buildAbstraction()` gets all the predicates, that are relevant at this location, from the predicate precision and identifies which of those still need to be handled by the Boolean abstraction. The lemmas for those predicates are selected by the `LemmaSelectionVisitor` and passed on to the method `computeAbstraction()`, together with the set of remaining predicates and the primary formula. `computeAbstraction()` initializes the prover environment with the primary formula and then passes the predicates and lemmas on to `computeBooleanAbstraction()` which does the actual computation of a Boolean abstraction.

To compute a Boolean abstraction, we first instantiate the symbolic atom of each predicate and build the equivalence between the symbolic atom and its symbolic variable. Then, we instantiate all the lemmas with the same variables as the predicates. The sets of the predicate equivalences and the instantiated lemmas are then added to the prover context. The prover performs a SAT check for all the remaining predicates that tells us whether the abstraction is satisfiable.

4.4 Selection of the Abstraction Lemmas

The `PredicatePrecision` contains one set of all lemmas from all the witness files. For the computation of a new abstraction, we want to select just the lemmas that are necessary for the abstraction formula and the relevant predicates. To achieve this, we use three visitors that implement the visitor pattern: The `LemmaSelectionVisitor`, the `LemmaVariableVisitor` and the `LemmaVariableEqualityVisitor`. CPACHECKER already provides the abstract class `DefaultFormulaVisitor` from which our three visitors inherit (see Fig. 4.4).

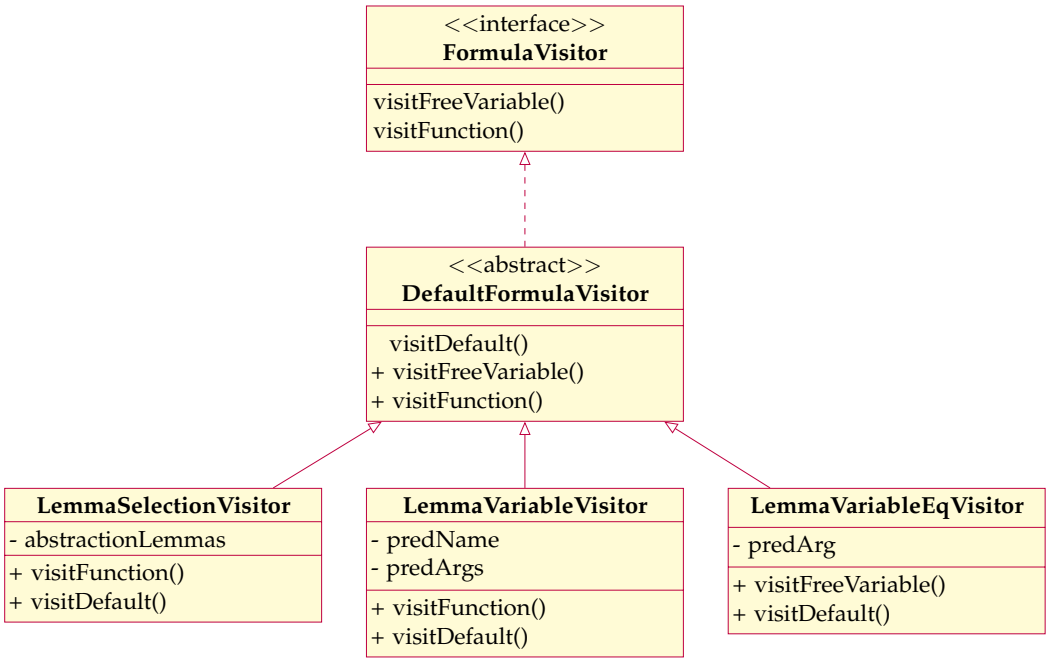


Figure 4.4: Lemma Selection Visitor

The `LemmaSelectionVisitor` has a set of all abstraction lemmas from which the relevant ones for a formula should be chosen. In our case, such a formula is either an abstraction formula or a predicate. In the default case, the visitor just returns the empty set. If it is visiting a function, the visitor will select all the lemmas for this function and then recursively traverse through its arguments.
If the name of the function is equal to the identifier of one of the lemmas, we have found a match. In this case, the formulas of the lemma need to be added to the set that the visitor will return. The variables in the definition of a lemma are never part of the program scope, while the arguments of a predicate are always from the program scope. For this reason, we need to add information about which lemma variable corresponds to which program variable to the resulting set. To find the relationship between the variables, we visit each formula from the matching lemma with the `LemmaVariableVisitor`.

To find the equality between a variable from the lemma and a variable from the formula or predicate, the `LemmaVariableVisitor` checks whether the name and number of arguments of the lemma match those of the formula. If this is the case, it will visit each argument of the formula with the `LemmaVariableEqualityVisitor`. The `LemmaVariableVisitor` returns a `BooleanFormula` of the form `<lemma argument> = <predicate argument>`, if it visits a free variable. Otherwise, it returns a trivially true `BooleanFormula`. The `BooleanFormula` that is returned by the `LemmaVariableEqualityVisitor` is then added to the result set of the `LemmaVariableVisitor`.

In the following Chapter 5, we evaluate our implementation for a selection of test programs, that cannot be verified by predicate abstraction in CPACHECKER without a lemma witness. Our approach dose have some limitations which we analyze in Chapter 6.

CHAPTER 5

Evaluation

To evaluate our implementation from Chapter 4 we use BENCHEXEC[1] [12, 13, 14], a framework that can produce reliable measurements for large sets of benchmarking tasks. For each benchmarking run it measures CPU time, wall time and memory usage while also allowing to set specific limits for these resources. We perform the benchmarking on a Debian 12 virtual machine with 2 CPU cores and 7 GiB of memory available on a host machine with an Intel i5-7Y54 CPU. The time limit we set for a benchmarking task is 900 s.

As input, BENCHEXEC takes a task definition file in the `XML` format and an input file in the `YAML` format. For each test program we create one input file that comes with a lemma witness where the expected result is `true` and another input file without a lemma witness. Without a witness the expected result is `UNKNOWN` because CPACHECKER reaches a `TIMEOUT`. For the benchmarking we use BENCHEXEC version 3.29 and CPACHECKER version 4.1-151-gee5d4a8bcd+.

In Sect. 5.1 of this chapter we evaluate our example Listing 1 from our *Background* Chapter 3, as well as a function that calculates the minimum from an array. In Sect. 5.2 we evaluate some examples from the `array-patterns` benchmarks that can be found in the `sv-benchmarks` [2] repository. In Sect. 5.3 we consider a modified version of one of the programs from Sect. 5.2 that shows the limitations of our approach. All program files that we use for benchmarking in this chapter have been added to the `cpachecker` [3] repository.

[1] `https://github.com/sosy-lab/benchexec`
[2] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks`
[3] `https://gitlab.com/sosy-lab/software/cpachecker/-/merge_requests/173`

5.1 Maximum and Minimum of an Array

In this section we take a closer look at the exemplary function `maxArray()` from Listing 3 in Chapter 3. The function loops over all elements from an array of integers and returns its maximum value. We want to verify that the call of `reach_error()` in line `11` cannot be reached when executing the program. When using predicate abstraction, CPACHECKER cannot verify this property without a lemma witness.

To verify that the variable `m` from `maxArray()` in Listing 3 is greater or equal to any element from the array we need the predicate `m == _L_MAX(arr,j)` as well as the supplementary lemmas `_L_MAX(A,0)== A[0]` and `_L_MAX(A,I) == A[I] > MAX(A,I-1) ? A[I] : MAX(A,I-1)`. The second lemma is easily readable and intuitive for humans if we write it as a ternary expression. However, the lemmas are parsed as a `CExpression` which does not implement ternary expressions. For this reason, we have to transform the second lemma of the witness from Listing 4 into a less intuitive binary expression that can be handled by CPACHECKER.

We want to validate that CPACHECKER can solve the verification task via predicate abstraction if we provide it with the lemma witness but that it cannot solve the same task without a lemma witness. Listing 5 shows the input file for an execution of CPACHECKER with a lemma witness for the `max-array()` function where the expected result is `true`. Listing 6 shows the input file for a run without a lemma witness where we expect a `TIMEOUT`.

Table 5.1 shows the results of a BENCHEXEC run for the functions `max-array()` and `min-array()` when CPACHECKER is provided with a lemma witness. As expected, both verification tasks return the result `true` within our given resource limits. Table 5.2 shows the results of a BENCHEXEC run for the same functions but without a lemma witness. In this case, the verification tasks cannot be solved within the given time frame and BENCHEXEC returns a `TIMEOUT`. These results show that the lemma witnesses are necessary for CPACHECKER to verify the `max-array()` and `min-array()` functions via predicate abstraction.

	status	**cputime (s)**	**walltime (s)**	**memory (MB)**
max-array	true	7.517857	4.890180	267.530240
min-array	true	9.110797	5.278691	259.887104

Table 5.1: Results for the BENCHEXEC run with a lemma witness

	status	**cputime (s)**	**walltime (s)**	**memory (MB)**
max-array	TIMEOUT	902.700347	884.923291	884.420608
min-array	TIMEOUT	903.079896	883.360306	1027.624960

Table 5.2: Results for the BENCHEXEC run without a lemma witness

Listing 3 A C function that returns the maximum from an array

```
int maxArray(int* a,unsigned int l, unsigned int i) {
  int j = 0;
  unsigned int m = a[0];
  while(j < l) {
    if(a[j] > m) {
      m = a[j];
    }
    j = j+1;
  }
  if(m < a[i]) {
    reach_error();
    return -1;
  }
  return m;
}
```

Listing 4 A lemma witness for the `maxArray()` function

```
- entry_type: invariant_set
  content:
    - invariant:
        type: loop_invariant
        value: "m == ACSL(_L_MAX(arr,j))"
        format: c_expression
- entry_type: lemma_set
  content:
    - lemma:
        value: "ACSL(_L_MAX(A,0)) == A[0]"
        format: c_expression
    - lemma:
        value: "((A[I] > ACSL(_L_MAX(A,I))) &
         ↪  ACSL(_L_MAX(A,I)) == A[I]) | (A[I] <=
         ↪  ACSL(_L_MAX(A,I)) & ACSL(_L_MAX(A,I)) ==
         ↪  ACSL(_L_MAX(A,I-1)))"
        format: c_expression
```

Listing 5 Input file with a witness

```
format_version: '2.1'

additional_information:
  task_type: validation
  verification:
    - property_file:
      ↪ ../../../config/properties/unreach-call.prp
      expected verdict: true

input_files:
  - 'max-array.c'
  - 'max-array-witness.yml'

properties:
  - property_file:
    ↪ ../../../config/properties/unreach-call.prp
    expected verdict: true

options:
  language: C
  data_model: ILP32
  witness_input_file: 'max-array-witness.yml'
```

Listing 6 Input file without a witness

```
format_version: '2.1'

additional_information:
  task_type: validation
  verification:
    - property_file:
      ↪  ../../../config/properties/unreach-call.prp
      expected verdict: unknown

input_files:
  - 'max-array.c'

properties:
  - property_file:
    ↪  ../../../config/properties/unreach-call.prp
    expected verdict: unknown

options:
  language: C
  data_model: produces ILP32produces
```

5.2 Array Patterns

In this section we evaluate a selection of `array-pattern` benchmarks from the `sv-benchmark` [4] repository. All of those programs follow a similar structure for which an excerpt from `array1_pattern.c` in Listing 7 and `array5_pattern.c` in Listing 8 give two examples:
First, we initialize two arrays of the same size with all values as zero. We then pick a random index from the first array and perform an operation that changes the value at that index. The value of a corresponding index from array two is then changed in a complementary way. At the end, we verify that the sum of both arrays fulfills some property, for example that the sum of both arrays adds up to exactly zero.

In `array1_pattern.c` we select a random index `i` between zero and the size of the array minus one. Then, we set the array entry at this index to the value of `i`. Next, we set the array entry `i` steps from the end of the array to the value of `-i`. After a non deterministic number of repetitions of this loop we calculate the sum of both arrays. As the last step, we verify that the sum of both arrays equals zero.
As we can see in Table 5.4, CPACHECKER cannot verify this property via predicate abstraction within our hardware restrictions. To verify that the sum `sum` of both arrays is equal to zero we need to provide CPACHECKER with the predicate `sum == _L_SUM(array1, array2, count)` from Listing 9 for the `for` loop in line `74-77`. The corresponding lemma `_L_SUM` is defined by the witness in Listing 9 as `_L_SUM(A1, A2, 0) == 0` and `_L_SUM(A1, A2, C) == _L_SUM(A1, A2, C-1) + A1[1] + A2[C]`.
If we provide CPACHECKER with this lemma witness, we can verify the program `array1_pattern.c` via predicate abstraction and within our constraints, as Table 5.3 shows.

The program `array5_pattern.c` in Listing 8 only differs from `array1_pattern.c` in how the values for the arrays are determined: For the first array, an even index `i` of the array is assigned the value of `i` while an odd index is assigned the value of `-i`. For the second array, an even index `i` of the array is assigned the value of `i` and an odd index is assigned the value of `i`.
Table 5.4 shows that we cannot verify `array5_pattern.c` via predicate abstraction, if we do not provide CPACHECKER with a predicate and lemma witness.

The calculation of the sum `sum` for `array5_pattern.c` in line `97-100` from Listing 8 is identical to the `for` loop line `74-77` from `array1_pattern.c`. This means that we can use the same predicate and lemma witness to verify both programs if we assign the predicate to the correct location in the source code. It would make sense to generate a new predicate witness for each program and reuse the same lemma witness file wherever it is applicable. But because of how the parsing of the lemma witnesses and the predicates that reference a lemma is currently implemented, all lemmas for a predicate need to be

[4] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks`

provided within the same witness file. As discussed in Sect. 7.1, the parsing of the lemmas and predicates should be adapted so that a lemma witness can be reused.

Because of the cumulative nature of the `array-pattern` benchmarks, we cannot verify them with predicate abstraction in CPACHECKER without a lemma witness, as Table 5.4 shows. If we provide CPACHECKER with a lemma witness that defines how the sum of two array is computed and a predicate that makes use of this lemma, CPACHECKER can verify the selected `array-pattern` programs within the given time frame via predicate abstraction. Table 5.3 gives the results for a BENCHEXEC run over the `array-pattern` benchmarks when CPACHECKER is provided with a lemma witness for predicate abstraction. As expected, in this case the verification result for all benchmarks is `true`.

	status	**cputime (s)**	**walltime (s)**	**memory (MB)**
array1_pattern	true	7.586583	4.474643	272.375808
array2_pattern	true	8.191312	4.913940	272.220160
array3_pattern	true	7.243493	4.043497	269.238272
array4_pattern	true	9.860977	5.642485	274.403328
array5_pattern	true	11.951999	7.337764	264.130560
array6_pattern	true	10.761578	5.943338	262.111232
array7_pattern	true	10.813531	6.523912	256.892928
array8_pattern	true	10.826994	6.643799	263.839744
array9_pattern	true	9.957992	5.786742	274.001920
array10_pattern	true	10.743326	6.628171	279.691264

Table 5.3: Results for the BENCHEXEC run with a lemma witness

	status	**cputime (s)**	**walltime (s)**	**memory (MB)**
array1_pattern	TIMEOUT	903.21816	883.372897	449.327104
array2_pattern	TIMEOUT	904.406783	885.379947	478.367744
array3_pattern	TIMEOUT	902.526338	885.938551	806.858752
array4_pattern	TIMEOUT	903.677846	883.405733	460.611584
array5_pattern	TIMEOUT	901.646286	883.473540	470.028288
array6_pattern	TIMEOUT	903.908666	885.320838	579.055616
array7_pattern	TIMEOUT	903.211315	883.882255	590.790656
array8_pattern	TIMEOUT	903.554847	882.603957	475.922432
array9_pattern	TIMEOUT	903.334656	884.473069	442.060800
array10_pattern	TIMEOUT	903.256535	883.766988	455.397376

Table 5.4: Results for the BENCHEXEC run without a lemma witness

Listing 7 Excerpt from `array1_pattern.c`

```
44 int main()
45 {
46         int ARR_SIZE = 10000;
47
48         int array1[10000] ;
49         int array2[10000] ;
50         int count = 0, num = -1 ;
51         int temp ;
52         short index ;
53         signed long long sum = 0 ;
54
55         for(count=0;count<ARR_SIZE;count++)
56         {
57                 array1[count] = 0 ;
58                 array2[count] = 0 ;
59         }
60
61         while(1)
62         {
63
64                 index = __VERIFIER_nondet_short() ;
65                 assume_abort_if_not(index>=0 && index <
                    ↪  ARR_SIZE) ;
66
67                 array1[index] = num*(num*index) ;
68                 array2[ARR_SIZE-1-index] = num * index ;
69
70                 temp = __VERIFIER_nondet_int() ;
71                 if(temp == 0) break ;
72         }
73
74         for(count=0;count<ARR_SIZE;count++)
75         {
76                 sum = sum + array1[count] + array2[count] ;
77         }
78
79         __VERIFIER_assert(sum == 0) ;
80         return 0 ;
81 }
82
```

Listing 8 Excerpt from `array5_pattern.c`

```
60          int count = 0, num = -1 ;
61             signed long long sum = 0 ;
62          int temp ;
63          short index ;
64
65          for(count=0;count<ARR_SIZE;count++)
66          {
67                  array1[count] = 0 ;
68                  array2[count] = 0 ;
69          }
70
71          count = 1 ;
72
73          while(1)
74          {
75
76                  index = __VERIFIER_nondet_short() ;
77                  assume_abort_if_not(index>=0 && index <
                     ↪  ARR_SIZE) ;
78
79
80                  if(index % 2 == 0){
81                          array1[index] = num * (num * count)
                             ↪  ;
82                          array2[index] = num * count ;
83                  }
84                  else{
85                          array1[index] = num * count ;
86                          array2[index] = num * (num * count)
                             ↪  ;
87                  }
88
89                  if(count == 200)
90                          count = 0 ;
91                  count++ ;
92
93                  temp = __VERIFIER_nondet_int() ;
94                  if(temp == 0) break ;
95          }
96
97          for(count=0;count<ARR_SIZE;count++)
98          {
99                  sum = sum  + array1[count] + array2[count];
100         }
101
102         __VERIFIER_assert(sum == 0) ;
```

Listing 9 Excerpt from a lemma witness for array1_pattern.c from Listing 7

```
- entry_type: invariant_set
  content:
    - invariant:
        type: loop_invariant
        location:
          line: 74
          column: 2
          function: main
        value: "sum == ACSL(_L_SUM(array1, array2, count))"
        format: c_expression
- entry_type: lemma_set
  declarations:
    - "int _L_SUM(int* A1, int* A2, int C)"
    - "int* A1"
    - "int* A2"
    - "int C"
  content:
    - lemma:
        value: "ACSL(_L_SUM(A1, A2, 0)) == 0"
        format: c_expression
    - lemma:
        value: "ACSL(_L_SUM(A1, A2, C)) == ACSL(_L_SUM(A1,
          ↪  A2, C-1)) + A1[C] + A2[C]"
        format: c_expression
```

5.3 Limitations

All examples in Sect. 5.1 and in Sect. 5.2 have the expected verdict `true` when provided with a lemma witness, and `TIMEOUT` otherwise. In this section we evaluate two modified versions of the program `array1_pattern.c` from Listing 7 of the previous section. We expect the modified program `array1_pattern-false.c` from Listing 10 to reach the error location upon execution. The program `array1_-pattern-always-false.c` has been modified in such a way, that the expected verdict is never `true`. Both modified programs were added to the `cpachecker` repository.

In line `79` from `array1_pattern.c` from Listing 7 we reach an error state if the sum of `array1` and `array2` is not equal to zero. Table 5.3 shows that the analysis of this program with a lemma witness yields the expected result `true`. If we invert this condition, we reach the error location if the sum of both array is equal to zero. An excerpt from the program `array1_pattern-false.c` that implements this change is shown in Listing 10.
Because predicate abstraction in CPACHECKER works with an over approximation of the state space, we expect a verification attempt for `array1_pattern-false.c` to reach a `TIMEOUT`. However, Table 5.5 shows that the actual result for a CPACHECKER run with the lemma witness from Listing 9 is `true`.

Listing 10 gives a program that adds an `else`-clause to the `if`-statement in line `79` of Listing 10. Because the `if`-clause and the `else`-clause both contain a `reach_error()` call, the program always reaches an error state upon execution. We expect a verification attempt of `array1_pattern-always-false.c` with predicate abstraction in CPACHECKER to reach a `TIMEOUT` or yield the verdict `false`. But, as we can see in Table 5.5, CPACHECKER still returns `true` when given this program with the lemma witness from Listing 9 for predicate abstraction.

We discuss possible solutions for this problem in Chapter 6.

array1_pattern-	**status**	**cputime (s)**	**walltime (s)**	**memory (MB)**
false	true	8.50163	5.272354	262.180864
always-false	true	8.116608	5.272354	5.272354

Table 5.5: BENCHEXEC run for Listing 10 and Listing 11 with a witness Listing 9

Listing 10 Excerpt from `array1_pattern-false.c`:
A version of Listing 7 where the property is error location is reachable

```
74          for(count=0;count<ARR_SIZE;count++)
75          {
76                  sum = sum+ array1[count] + array2[count] ;
77          }
78
79          if(sum == 0){
80                  reach_error();
81                  return -1;
82          }
83
84          return 0 ;
85  }
```

Listing 11 Excerpt from `array1_pattern-always-false.c`:
A version of Listing 7 where an error location will always be reached

```
74          for(count=0;count<ARR_SIZE;count++)
75          {
76                  sum = sum+ array1[count] + array2[count] ;
77          }
78
79          if(sum == 0){
80                  reach_error();
81                  return -1;
82          }
83          else{reach_error();}
84
85          return 0 ;
```

CHAPTER 6

Discussing Current Limitations

During the evaluation in Chapter 5, we discussed how CPACHECKER falsely returns the verification result `true` for programs that should reach an error at runtime. In Sect. 5.3 we have evaluated one program in Listing 10 where the error location should be reachable and another program in Listing 11 that should always reach an error location during execution. But the verdict for both of these programs for predicate abstraction with a lemma witness is `true`.
In Chapter 3 we have seen how predicate abstraction in CPACHECKER produces an abstract reachability graph (ARG) that maps abstract states to concrete locations in the CFA. This way CPACHECKER determines if a program location is reachable at runtime.

When CPACHECKER analyses both programs, it unrolls the ARGs as expected until the loop head in line `74`. It also reaches the summation `sum = sum + array1[count] + array2[count]` of the loop-body in line `76`. After this abstract state both ARGs are not unrolled further, even though we would expect them to go back to the loop-head in line `74`.
The ARGs for both programs reach the `if`-statement `if(sum == 0)` in line `79`. From here, we should reach the `reach_error()`-call in line `80`, but the ARGs do not contain an abstract state that covers this program location.
In the ARG for `array1_pattern-false` from Listing 10 we have an abstract state for the `return`-statement in line `84`.
With the program from Listing 11 we expect to reach an error location in any case after line `79`. But the ARG for this program does not unroll further after this point.

During the execution of the CPA Algorithm (Algorithm 1), the *predicate transfer relation* $\rightsquigarrow_{\mathbb{P}}$ from Definition 1 calculates all abstract successors for a given abstract state. Afterwards, the stop operator $stop_{\mathbb{P}}$ from Definition 3 determines, whether those abstract successors are implied by another abstract state that is already part of the waitlist. In this case, the abstract successors do not need to be explored further.

We have discussed how the ARGs for are not fully explored at the loop-head in line `74` and at the error locations in lines `80` and `84`. It appears that CPACHECKER does not recognize these program locations as abstraction locations.
One possible cause for this behavior is that the *predicate transfer relation* $\rightsquigarrow_{\mathbb{P}}$ does not find any abstract states corresponding to these program locations. It is also possible that the *predicate transfer relation* finds the correct abstract successors, but that the stop operator falsely decides that they are implied by another abstract state and should not be explored further.

We suggest to investigate why the ARGs for Listing 10 and Listing 11 are not explored fully at the relevant abstraction locations. Based on the analysis given in this chapter, we propose that the *predicate transfer relation* $\rightsquigarrow_{\mathbb{P}}$ or the stop operator $stop_{\mathbb{P}}$ is likely the cause for the limitations that we have discussed in Sect. 5.3.

CHAPTER 7

Future Work

With the implementation proposed by this thesis we introduce a prototype that can can solve the Boolean predicate abstraction for loops with cumulative properties over arrays. In Chapter 5 we have evaluated our approach and discovered that lemmas cannot always be expressed in the most intuitive way. We have also seen that the same lemma witness cannot easily be reused for every program where it is applicable. Sect. 7.1 suggests to integrate a dedicated ACSL-Parser which would mitigate those limitations. In Sect. 3.2 we suggest to extract the functionality of our prototype into a new *Higher Order Abstraction CPA*.

7.1 Integrating an ACSL Parser

In Sect. 4.2, we have discussed how lemmas are parsed into CPACHECKER by using the existing infrastructure around `CExpressions`. This approach, however, has the limitation that we can only parse expressions that are covered by `CExpressions`. Function calls, for example, are not part of `CExpression` and could not easily be parsed without the introduction of the new class `ACSLFunctionCall`.

As part of the evaluation in Chapter 5 we discussed that `CExpression` does not support ternary expressions. Because of this limitation we had to represent an intuitive and easily readable ternary expression as a much more complicated binary expression. In Chapter 5 we also noticed that we cannot reuse the same lemma witness for multiple predicate witnesses, because we need the lemma declarations for the parsing of a predicate that references a lemma.

Because of the limitations of `CExpressions` in CPACHECKER, a lemma witness should be represented as an `acsl_expression` instead of a `c_expression`. To parse these expressions an ACSL parser that can handle the types of expressions that are

relevant for the lemmas should be integrated into the current approach.

For this we would use the interface `ACSLExpression` that exists in parallel to the interface `CExpression`. The new interface and the corresponding parser would then also be able to handle function calls and ternary expressions.
For the interface `ACSLExpression` we would also need a `ToFormulaVisitor` that could transform an `ACSLExpression` into a `BooleanFormula` which can be used for the abstraction.

There are also changes that can be made to the data structure into which the lemmas are parsed. We make a proposal for such changes in the following section.

7.2 Higher Order Abstraction CPA

With our current implementation a set of global abstraction lemmas is part of the `PredicatePrecision`. During each abstraction step and for every relevant predicate, we iterate over the whole set of lemmas to select the ones that apply at the current location. Instead, we could introduce a *Higher Order Abstraction CPA* which would have a lemma precision in analogy to the predicate precision with global, local and function specific lemmas. Like the current `PredicatePrecision`, a `LemmaPrecision` could be refined.

CHAPTER 8

Conclusion

The current state of predicate abstraction in CPACHECKER can verify some programs containing loops with user defined predicates, but not those that operate on cumulative properties over arrays. We expanded the implementation of predicate abstraction in CPACHECKER to successfully compute the Boolean predicate abstraction for functions that loop over arrays. Interesting properties of arrays are finding the maximum or minimum element from an array or comparing the sums of to arrays.
Our approach makes it possible to compute the Boolean predicate abstraction for cumulative array functions by supplementing the user defined predicates with also user defined lemmas. Those lemmas are provided to CPACHECKER via a witness file and then parsed into CPACHECKER as a `CExpression`. The current implementation of C expressions in CPACHECKER does not support function calls or ternary expressions which caused difficulties for the parsing of the lemmas. For future work we propose to introduce the new interface `ACSLExpression` and a corresponding parser which provide all the necessary functionality.
Within CPACHECKER the lemmas are a part of the `PredicatePrecision`. For every abstraction step we select the lemmas that supplement the relevant predicates for this location and add them to the solver context. This makes it possible for the SMT-Solver to successfully solve the verification task. The current implementation of the lemmas as part of the `PredicatePrecision` is not yet optimized. For future work we propose to extract the functionality of the lemmas into a *Higher Order Abstraction CPA* that has a more refined `LemmaPrecision`. This new CPA would then interact with the Predicate CPA to solve a verification task.
We have shown that the Boolean predicate abstraction for loops over arrays can be computed by CPACHECKER using predicate abstraction. To achieve this goal we supplement user defined predicates with user defined lemmas. This way we can use more inductive predicates which enable us to compute the Boolean predicate abstraction for cumulative functions like calculating the maximum from an array of integers.

Bibliography

[1] W. Ahrendt, B. Beckert, R. Bubel, R. H ähnle, P. H. Schmitt, and M. Ulbrich. *Deductive software verification-the key book*. Springer, 2016. DOI: 10.1007/978-3-319-49812-6.

[2] J. Amilon, Z. Esen, D. Gurov, C. Lidström, and P. Rümmer. Automatic program instrumentation for automatic verification. In *CAV*, pages 281–304. Springer, 2023. DOI: 10.1007/978-3-031-37709-9_14.

[3] J. Amilon, Z. Esen, D. Gurov, C. Lidström, and P. Rümmer. An exercise in mind reading: Automatic contract inference for frama-c. In *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications*, pages 553–582. Springer, 2024. DOI: 10.1007/978-3-031-55608-1_13.

[4] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, 2001. DOI: 10.1145/378795.378846.

[5] T. Ball and S. K. Rajamani. The slam toolkit. In *Proceedings of CAV'2001*, volume 2102, pages 260–264, 2000.

[6] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, 2002. DOI: 10.1145/503272.503274.

[7] P. Baudin, F. Bobot, L. Correnson, Z. Dargaye, and A. Blanchard. Wp plug-in manual. *Frama-c. com*, 2020.

[8] D. Beyer, M. Dangl, and P. Wendler. A unifying view on smt-based software verification. *Journal of automated reasoning*, 60(3):299–335, 2018. DOI: 10.1007/s10817-017-9432-6.

[9] D. Beyer, S. Gulwani, and D. A. Schmidt. Combining model checking and data-flow analysis. *Handbook of Model Checking*, pages 493–540, 2018. DOI: 10.1007/978-3-319-10575-8_16.

[10] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast: Applications to software engineering. *STTT*, 9(5):505–525, 2007. DOI: 10.1007/s10009-007-0044-z.

[11] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 184–190. Springer, 2011. DOI: 10.1007/978-3-642-22110-1_16.

[12] D. Beyer, S. Löwe, and P. Wendler. Benchmarking and resource measurement. In *SPIN*, pages 160–178. Springer, 2015. DOI: 10.1007/978-3-319-23404-5_12.

[13] D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: requirements and solutions. *STTT*, 21(1):1–29, 2019. DOI: 10.1007/s10009-017-0469-y.

[14] D. Beyer and P. Wendler. Cpu energy meter: A tool for energy-aware algorithms engineering. In *TACAS*, pages 126–133. Springer, 2020. DOI: 10.1007/978-3-030-45237-7_8.

[15] R. V. Book and F. Otto. String-rewriting systems. In *String-Rewriting Systems*, pages 35–64. Springer, 1993. DOI: 10.1007/978-1-4613-9771-7_3.

[16] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI 2006, Charleston, SC, USA, January 8-10, 2006. Proceedings 7*, pages 427–442. Springer, 2006. DOI: 10.1007/11609773_28.

[17] M. Broy, M. V. Cengarle, and E. Geisberger. Cyber-physical systems: imminent challenges. In *Monterey workshop*, pages 1–28. Springer, 2012. DOI: 10.1007/978-3-642-34059-8_1.

[18] K. Brunnstein. About the "altona railway software glitch". *The Risks Digest*, 16:93, 1995.

[19] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50(5):752–794, 2003. DOI: 10.1145/876638.876643.

[20] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *TACAS*, pages 168–176. Springer, 2004. DOI: 10.1007/978-3-540-24730-2_15.

[21] L. Cordeiro, J. Morse, D. Nicole, and B. Fischer. Context-bounded model checking with esbmc 1.17: (competition contribution). In *TACAS*, pages 534–537. Springer, 2012. DOI: 10.1007/978-3-642-28756-5_42.

[22] L. Correnson, P. Cuoq, F. Kirchner, A. Maroneze, V. Prevosto, A. Puccetti, J. Signoles, and B. Yakobowski. Frama-c user manual. *CEA LIST*, 111, 2010.

[23] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. In *SEFM*, pages 233–247. Springer, 2012. DOI: 10.1007/978-3-642-33826-7_16.

[24] P. Daca, T. A. Henzinger, and A. Kupriyanov. Array folds logic. In *CAV*, pages 230–248. Springer, 2016. DOI: 10.1007/978-3-319-41540-6_13.

[25] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11*, pages 160–171. Springer, 1999. DOI: 10.1007/3-540-48683-6_16.

[26] E. De Angelis, F. Fioravanti, J. P. Gallagher, M. V. Hermenegildo, A. Pettorossi, and M. Proietti. Analysis and transformation of constrained horn clauses for program verification. *Theory and Practice of Logic Programming*, 22(6):974–1042, 2022. DOI: 10.1017/S1471068421000211.

[27] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. DOI: 10.1145/360933.360975.

[28] E. W. Dijkstra, E. W. Dijkstra, E. W. Dijkstra, E.-U. Informaticien, and E. W. Dijkstra. *A discipline of programming*, volume 613924118. prentice-hall Englewood Cliffs, 1976.

[29] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978. DOI: 10.1145/359642.359655.

[30] E. W. Dijkstra and C. S. Schölten. The strongest postcondition. In *Predicate Calculus and Program Semantics*, pages 209–215. Springer, 1990. DOI: 10.1007/978-1-4612-3228-5_12.

[31] G. Ernst. Korn—software verification with horn clauses (competition contribution). In *TACAS*, pages 559–564. Springer, 2023. DOI: 10.1007/978-3-031-30820-8_36.

[32] Z. Esen and P. Rümmer. Tricera verifying c programs using the theory of heaps. In *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design*, pages 360–391. TU Wien Academic Press, 2022.

[33] R. W. Floyd. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*, pages 65–81. Springer, 1993. DOI: 10.1007/978-94-011-1793-7_4.

[34] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *CAV'97 Haifa, Israel, June 22–25, 1997 Proceedings 9*, pages 72–83. Springer, 1997. DOI: 10.1007/3-540-63166-6_10.

[35] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In *CAV*, pages 343–361. Springer, 2015. DOI: 10.1007/978-3-319-21690-4_20.

[36] P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about integer arrays? In *FoSSaCS*, pages 474–489. Springer, 2008. DOI: 10.1007/978-3-540-78499-9_33.

[37] R. Hähnle and M. Huisman. Deductive software verification: from pen-and-paper proofs to industrial tools. *Computing and Software Science: State of the Art and Perspectives*, pages 345–373, 2019. DOI: 10.1007/978-3-319-91908-9_18.

[38] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, 2002. DOI: 10.1145/503272.503279.

[39] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. DOI: 10.1145/363235.363259.

[40] C.-D. Hong and A. W. Lin. Regular abstractions for array systems. *Proceedings of the ACM on Programming Languages*, 8(POPL):638–666, 2024. DOI: 10.1145/3632864.

[41] M. Huisman, W. Ahrendt, D. Grahl, and M. Hentschel. Formal specification with the java modeling language. In *Deductive Software Verification–The KeY Book: From Theory to Practice*, pages 193–241. Springer, 2016. DOI: 10.1007/978-3-319-49812-6.

[42] T. Kahsai, R. Kersten, P. Rümmer, and M. Schäf. Quantified heap invariants for object-oriented programs. In *LPAR-21, May 7–12, 2017, Maun, Botswana*, pages 368–384, 2017.

[43] T. Kahsai and C. Tinelli. Pkind: A parallel k-induction based model checker. *arXiv preprint arXiv:1111.0372*, 2011. DOI: 10.48550/arXiv.1111.0372.

[44] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, pages 267–281. Springer, 2004. DOI: 10.1007/978-3-540-24622-0_22.

[45] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, et al. Jml reference manual, 2008. DOI: 10.1007/978-3-319-49812-6.

[46] K. R. M. Leino and V. Wüstholz. The dafny integrated development environment. *arXiv preprint arXiv:1404.6602*, 2014. DOI: 10.4204/EPTCS.149.2.

[47] D. C. Luckham and F. W. Von Henke. An overview of anna, a specification language for ada. *IEEE Software*, 2(2):9, 1985. DOI: 10.1109/MS.1985.230345.

[48] Y. Matsushita, T. Tsukada, and N. Kobayashi. Rusthorn: Chc-based verification for rust programs. *ACM TOPLAS*, 43(4):1–54, 2021. DOI: 10.1145/3462205.

[49] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136. Springer, 2006. DOI: 10.1007/11817963_14.

[50] B. Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 2002. DOI: 10.1109/2.161279.

[51] Z. Rakamarić and M. Emmi. Smack: Decoupling source language details from verifier implementations. In *CAV*, pages 106–113. Springer, 2014. DOI: 10.1007/978-3-319-08867-9_7.

[52] D. M. Ritchie, S. C. Johnson, M. Lesk, B. Kernighan, et al. The c programming language. *Bell Sys. Tech. J*, 57(6):1991–2019, 1978.

[53] C. Sinz, F. Merz, and S. Falke. Llbmc: A bounded model checker for llvm's intermediate representation: (competition contribution). In *TACAS*, pages 542–544. Springer, 2012. DOI: 10.1007/978-3-642-28756-5_44.

[54] N. Suzuki and D. Jefferson. Verification decidability of presburger array programs. *JACM*, 27(1):191–205, 1980.

[55] T. Wahl. The k-induction principle. *Northeastern University, College of Computer and Information Science*, pages 1–2, 2013.

[56] P. Wendler. *Towards Practical Predicate Analysis*. PhD thesis, 11 2017.

[57] M. Wenzel, L. C. Paulson, and T. Nipkow. The isabelle framework. In *TPHOLS*, pages 33–38. Springer, 2008. DOI: 10.1007/978-3-540-71067-7_7.