Ludwig Maximilian University of Munich
Institute of Informatics
Software and Computational Systems Lab

Bachelor's Thesis
in Computer Science plus Mathematics

SMT-based Model Checking of Concurrent Programs

Vladyslav Kolesnykov

Supervisor:	Prof. Dr. Dirk Beyer
Mentor:	Karlheinz Friedberger
Date:	26.11.2020

Eigenständigkeitserklärung

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

München, den 26.11.2020

. . . .
Vladyslav Kolesnykov

Acknowledgement

I would like to thank all the teaching staff at the Ludwig Maximilian University Institute of Informatics for their excellent teaching and the valuable guidance they have given me throughout my studies. They provided me with the tools that I needed to choose the right direction in my studies and to successfully complete an important stage of my life.
I would like to especially thank my mentor, Karlheinz Friedberger, who always provided me with excellent support and assistance during my research. His knowledge and feedback pushed me to sharpen my thinking and brought my work to a higher level. I also wish to express my sincere thanks to Professor Beyer for his trust and support during my research and for his help in my application to a master's degree program in Software Engineering.
I am also grateful to my first school teacher in informatics, Eugene Moturnak, who recognized my potential and instilled a love of this subject in me. His motivation and teaching have helped me to determine my future life path.

Abstract

Nowadays, modern software applications are complex concurrent and distributed software systems that should be highly reliable and efficient, without data races, deadlocks, and other program bugs. Thus, the automated verification of concurrent programs is becoming increasingly important in order to benefit from the potential of advanced multi-core hardware and distributed infrastructure. However, this process can be challenging even for modern verification software due to the non-deterministic behavior of multithreaded programs. Several successful automated software verification approaches are based on the Satisfiability Modulo Theories (SMT), solving first-order-logic formulas over predicates. This thesis studies two SMT-based software verification techniques, Bounded Model Checking and Predicate Abstraction, as well as a configurable verification framework CPAchecker for C programs. Our framework implements these approaches in a single configurable component for predicate-based analyses, representing Bounded Model Checking and Predicate Abstraction in a single setting. In addition to the theoretical contribution, we present our implementation that extends the existing components of CPAchecker to use Bounded Model Checking and Predicate Abstraction for concurrent programs. The predicate-based analyses component is used with the underlying reachability analysis that explores the program's state-space analyzing all possible thread interleavings. We evaluate the performance of implemented verification techniques and some optimizations on the broad set of concurrent benchmark tasks, comparing them with other existing analyses in the same framework CPAchecker. We also present a combination of our techniques with explicit value-analysis to solve the state-space explosion problem and achieve even better verification performance. Finally, the implemented changes are applied as part of CPAchecker to participate in the International Competition on Software Verification 2021.

Contents

Chapter 1

Introduction

Considerable progress in the development of techniques for automatic formal verification in recent decades has increased interest in software verification of complex industrial systems. Therefore, leading hardware and software companies have started to integrate them into their products' quality assurance processes. Model Checking is one of the most widely used approaches, which attempts to verify whether a finite-state model of a system satisfies a provided specification formalized by temporal logic. Generally, there are two types of specification properties: safety properties (indicates what should not happen) and liveness properties (indicates what should finally happen). The process of proving or disproving the model's correctness involves exploring its all reachable states and transitions. If no program state that violates property is found, the model is proven to be correct. Otherwise, a sequence of the model's states called counterexample is generated to confirm the property contradiction.

In order to prove whether the finite-state model meets a given specification, the first model checking algorithms explicitly enumerated the reachable states. Therefore, the performance of such approaches was limited since the state-space can grow exponentially. The first model checkers could only verify models with a few million states, which prevented scaling up to the industrial complex. However, the introduction of symbolic model checking led to wide usage of this technique that allows representing a state space of a model implicitly using Boolean functions. Thus, the model can be traversed more efficiently by manipulating Boolean functions that enables handling the large numbers of states in a single step. The first symbolic methods used binary decision diagrams (BDDs), a data-structure for the representation of Boolean functions. The symbolic model checking with BDDs allowed verifying the system with an even larger set of states. Still, it required a considerable amount of memory for storing and manipulating BDDs. In addition, the variable ordering in the BDD directly affected the execution time of the model checker.

Due to impressive progress in the field of satisfiability theories (SAT) and satisfiability modulo theories (SMT) in recent decades, symbolic model checking has achieved considerable success. The state-space reachability problem can be reduced to propositional satisfiability (SAT) or generalizations thereof (SMT) and be solved by SAT or SMT solvers rather than BDDs. Consequently, the significant improvement in performance and scalability of Boolean satisfiability solvers resulted in their wide usage in symbolic model checking. The new generation of satisfiability solvers is able to handle propositional satisfiability problems with hundreds of thousands of variables[BCC$^+$03].

This thesis focuses on the two Symbolic Model Checking approaches, Bounded Model Checking (BMC) and Predicate Abstraction, which are both based on SMT solving as the back-end technology. BMC unrolls all program paths in an SMT formula and attempts to prove whether it contains a feasible program path leading to a violation of the specification. Another widely used technique is Predicate Abstraction, which simplifies and proves the properties of the system. It is usually applied with counterexample guided abstraction refinement (CEGAR) that refines the abstracted models iteratively using information obtained from counterexamples. This process continues until either property contradiction is found or proven that the model satisfies the given specification. The Bounded Model Checking and Predicate Abstraction methods are already implemented in the configurable software verification framework CPAchecker and competitive in verifying a large benchmark set of sequential programs.

Due to the increasing importance in the verification of concurrent programs, the main goal of this work is to extend the configurable verification framework CPAchecker with the support of Bounded Model Checking (BMC) and Predicate Abstraction for multithreaded programs and compare them with other existing software verification approaches. However, the analysis of multithreaded programs has a higher level of complexity. Multithreading enables the execution of multiple program tasks simultaneously and switching between the running processes. The number of program paths to be explored can grow exponentially with the program's length due to all possible interleaving of threads. This behavior causes the path explosion problem of symbolic execution and prevents verification techniques from scaling to be able to handle complex systems. However, by extending and optimizing some existing components of the CPAckecker, we achieved sound and efficient model checking of multithreaded programs.

The rest of the thesis is structured as follows. The next section gives a technical introduction to the verification framework CPAckecker and symbolic SMT-based verification approaches. Section 3 describes the implemented changes and optimizations in components of CPAchekcer for verification of multithreaded programs. In Section 4, we present the Bounded Model Checking and Predicate Abstraction evaluation and compare them with other existing software verification approaches for concurrent programs. Finally, the paper's results and possible directions for future work are discussed in the conclusion.

Chapter 2

Background

The following section provides an overview of some fundamental concepts and definitions used for SMT-based verification of concurrent programs. Additionally, we describe the main components of the software verification framework CPAchecker and details of configurable program analysis.

2.1 CPAchecker

CPAchecker is an open-source framework for configurable software verification and program analysis of C programs, which has been developed by the members of the chair for Software and Computational Systems at the Ludwig Maximilian University of Munich. CPAchecker has proven to be a successful tool for automatic software verification, annually winning a series of medals in different categories at the International Competition on Software Verification[Bey20].

CPAchecker is based on the concept of configurable program analysis (CPA), which provides the expressing different verification approaches using a single formal setting. It enables flexible and customizable program verification combining two major techniques of program analysis and model checking. The main design principle of CPAchecker is the separation of concerns that allows to separate various tasks, which are required for program verification, into independent components (CPAs). For instance, LocationCPA tracks program counter, CallstackCPA represents a program call stack, ThreadingCPA handles multiple program threads, etc. The concept of separation of concern also is applied to the different approaches of program analysis. Thus, the predicate and value analyses are implemented as individual CPAs: PredicateCPA and ValueCPA. In addition, CPAchecker offers interfaces to various SMT solvers such as

MathSAT5[CGSS13] and SMTInterpol[CJA12] for solving and interpolating over SMT formulas [BEK11, Löw17]. Figure 2.1 illustrates the coarse architecture of CPAchecker, displaying its most relevant components.

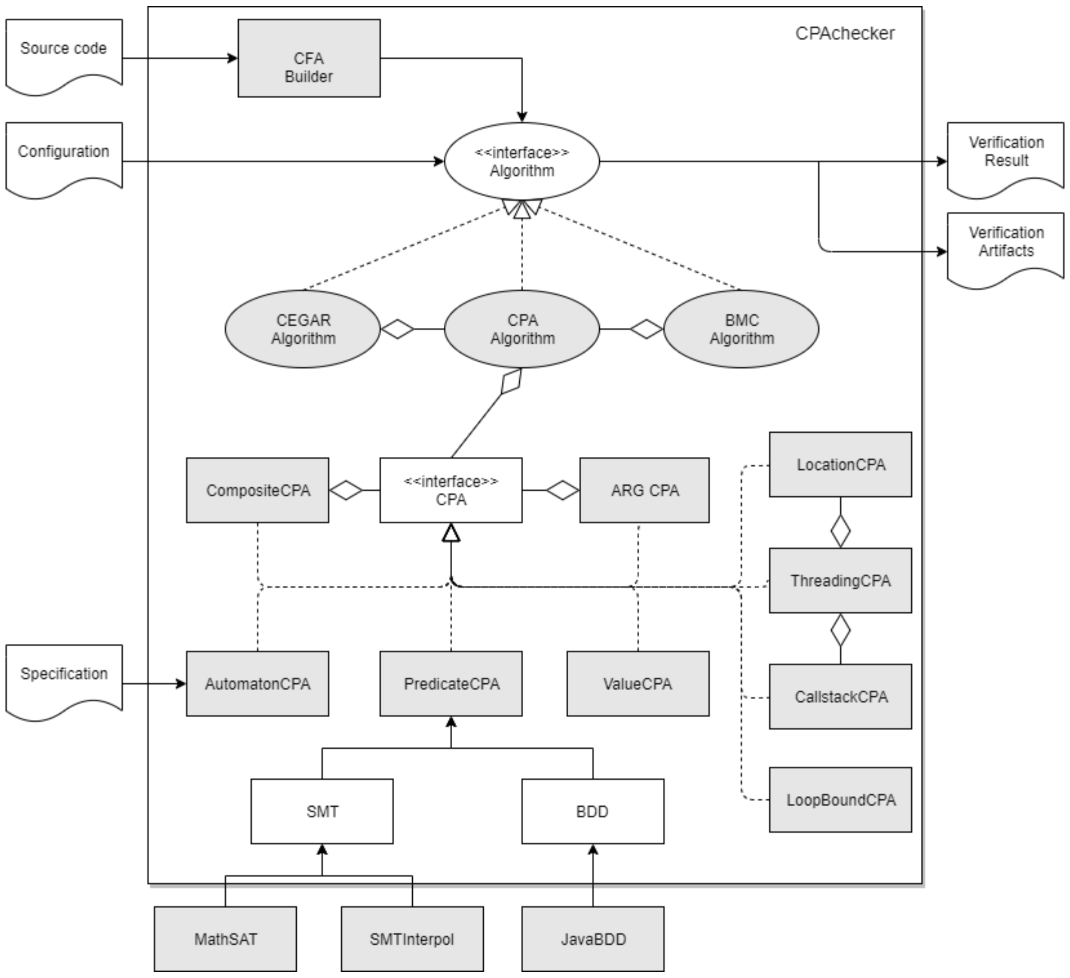

Figure 2.1: Architecture of the CPAchecker framework.

CPAchekcer accepts a program source code written in C and the specification, which contains specific properties that the program should fulfill. Firstly, the framework uses the C parser to transform the program into a control-flow automaton (CFA), an intermediate program representation. Then CPAchecker runs the CPA algorithm to perform reachability analysis, which accepts a constructed CFA and a set of CPAs required for a specific program analysis type. Finally, other algorithms, like CEGAR or BMC algorithm, use the program's state-space explored by the underlying CPA algorithm and provide the program verification. If a state, which violates a given specification, can be reached during a specific analysis, CPAchecker generates a counterexample and reports the program unsafe. Otherwise, the program meets the given specification.

In the following section, we describe each component of the CPAchecker in more detail.

2.2 Control Flow Automaton

Before the CPAchecker runs a specific program analysis, it parses the source code and constructs a control-flow automaton (CFA). CFA is a directed graph in which nodes denote a current program location that models the program counter. Its edges represent program statements that perform a transition from one program state to another. This program representation allows the traversal of all possible paths during the execution of verification analysis.

Formally, a CFA is a triple (L, l_0, G) that consists of the finite set L of all program locations, the start location $l_0 \in L$, and set $G \subseteq L \times Ops \times L$ of edges between program locations. Each edge $g \in G$ is denoted $l_i \xrightarrow{\text{op}} l_j$ with an operation executed when the control flows from a predecessor location l_i to a successor location l_j. An operation $op \in Ops$ can be an assignment of from $v := e$ where $v \in V$ is a variable and e is an arithmetic expression, or an assume operation $[p]$ with a predicate p over a set V.

The sequence $p = (l_0, op_0, l_1), (l_1, op_1, l_2), ..., (l_n, op_n, l_{\text{n+1}})$ of successive edges from G that starts at initial location l_0 is called a program path. The path p is feasible if there exists an execution order of program statements (edges) that leads to a specific program location. If there is a feasible program path p from location l_0 to l_n, a location l_n is called reachable.

Let $l_{\text{err}} \in L$ be an error location in CFA $A = (L, l_0, G)$. The primary purpose of software verification is either to prove that l_{err} is unreachable in A or to find a feasible program path to the error location.[BDW17]

2.3 Configurable Program Analysis

CPAchecker fulfills the principle of configurable program analysis, where various components (denoted as CPAs) are responsible for analyzing different program aspects. Later, CPAchecker was extended with the concept of dynamic precision adjustment that was introduced by Beyer, Henzinger, and Théoduloz in [BHT07]. This extension allows CPAchecker to adjust the precision of its components (CPAs) on the fly-way during the program verification. Thus, the component program analysis during execution can be configured either to be more abstract or precise and efficient or more precise but expensive.

According to the definition taken from [BHT08], a **CPA** $D = (D, \Pi, \rightsquigarrow, merge, stop, prec)$ consists of an abstract domain D, a set Π of precisions, a transfer relation $\rightsquigarrow$, a merge operator *merge*, a termination check operator *stop*, and a precision adjustment function *prec*.

Let V denote the finite set of program variables. A concrete state $(s, l) : (V \to \mathbb{Z}) \times L$ is a tuple of a program location $l \in L$ and a concrete data state $s \in (V \to \mathbb{Z})$ that assigns an integer value to each variable from the set V.

The **abstract domain** $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set C of concrete states, a semilattice $\mathcal{E} = (E, \sqsubseteq)$ and a concretization function $[\![\cdot]\!]$. A semilattice $\mathcal{E} = (E, \sqsubseteq)$ is a set E of abstract states with a partial order $\sqsubseteq$. A concretization function $[\![\cdot]\!]$ assigns to each abstract state $e \in E$ the set of concrete states that it represents.

The set Π of **precisions** defines the precisions of the abstract domain. The program analysis keeps track of different precisions for different abstract states using the precisions from Π.

The **transfer relation** $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ computes all successor $e' \in E$ for abstract state $e \in E$ under a precision Π. We denote $e \overset{g}{\rightsquigarrow} e'$ if there exists the edge $g \in G$ in the CFA between abstract states e and e'.

The **merge operator** $Emerge :\subseteq E \subseteq \Pi \rightsquigarrow E$ combines the information of two abstract states when the control flow meets under a given precision.

The **stop operator** $stop : E \times 2^E \times \Pi \rightsquigarrow B$ is also called termination check and used for coverage checks. It determines whether an abstract state $e \in E$ is already covered by a given set of abstract states $R \in 2^E$. If it is the case, then the stop operator returns true, and the CPA algorithm does not process the successor states of abstract state e. Merging abstract states and coverage checks significantly reduces the number of abstract states and program paths, avoiding state space explosion.

The **precision-adjustment operator** $prec : E \times \Pi \times 2^E \times \Pi \to E \times \Pi$ allows adjusting the analysis precision dynamically, increasing or decreasing the precision of abstract states. It computes a new abstract state and precision depending on the current set of reachable abstract states.

The concept of configurable program analysis also allows us to separate common analysis components into different CPAs. Thus, they should not be redefined for every analysis and can be efficiently reused. Several CPAs can be combined by CompositeCPA and used together for eliminating infeasible paths during the program analysis. The abstract state and precisions of the CompositeCPA are tuples of abstract states and precisions from each component of CompositeCPA. The operators *merge*, *stop*, *prec* of the CompositeCPA delegate the execution to the corresponding operators of component CPAs.

Different CPAs analyze different program aspects and thus might be able to prove and eliminate different program paths. The program analysis finds a specific path feasible if all components of the CompositeCPA agree with its feasibility. CPAs do not necessarily have to know about each other. However, the precision adjustment operator of the CompositeCPA supports the exchange of information between the component CPAs to achieve higher precision. The composite precision adjustment operator can adjust the precision of the component CPAs individually. Thus, it can increase the precision of one component analysis and decrease the precision of another simultaneously.

2.3.1 ThreadingCPA

For the analysis of single-threaded programs, the CPAchecker uses the LocationCPA and the CallstackCPA in order to track the program location and call stacks containing function calls with respective return location in the CFA. Unfortunately, those CPAs do not support the analysis of multithreaded programs. Thus, the ThreadingCPA was created to be able to explore the state space of the multithreaded program. The ThreadingCPA is a replacement for the LocationCPA and the CallstackCPA that can handle multiple program locations per abstract state with their call stacks.

The definition of the ThreadingCPA was taken from [BF16]. The **ThreadingCPA** $T = (D_\mathrm{T}, \rightsquigarrow_\mathrm{T}, merge_\mathrm{T}, stop_\mathrm{T})$ conforms to the structure of configurable program analysis.

The **abstract domain** $D = (C, \mathcal{J}, [\![\cdot]\!])$ consists of the set C of concrete states, the flat semi-lattice $\mathcal{J} = (E, \sqsubseteq, \sqcup, \bot)$, and the concretization function $[\![\cdot]\!] : \mathcal{J} \rightarrow 2^C$. Let I be the set of all possible thread identifiers. Each abstract threading state $e \in E$ consists of assignments $\{t_1 \rightarrow l^{t_1}, t_2 \rightarrow l^{t_2}, ..., t_n \rightarrow l^{t_n}\}$ of thread identifier $t \in I$ to current program location $l^{t_j} \in L \cup \{\top_\mathrm{L}\}$. $\top_\mathrm{L}$ represents an unknown program location. A semilattice $\mathcal{J} = (E, \sqsubseteq, \sqcup, \bot)$ is a set E of abstract states with a partial order $\sqsubseteq$. The join operator $\sqcup$ yields the least upper bound of given abstract states. $\bot = \sqcup E$ is the top element of the semi-lattice $\mathcal{J}$.

The **merge operator** $merge_\mathrm{sep}$ of the ThreadingCPA does not combine different threading abstract states.

The **stop operator** $stop_\mathrm{sep}$ of the ThreadingCPA determines coverage only if both threading abstract states are equal.

The **precision adjustment operator** *prec* never changes the precision of threading abstract state.

The **transfer relation** $\rightsquigarrow_\mathrm{T}$ of ThreadingCPA computes all possible successors for all active threads in the current abstract state. It uses the transfer relation of the LocationCPA for each active thread. Moreover, transfer relation is able to handle thread-related function calls (*pthread_create* and *pthread_join*) that change either the number of threads or the progress of thread executions. For example, the transfer relation $\rightsquigarrow_{\mathrm{T}}$ processes the CFA edge $g = (l^{t_j}, op, l^{t'_j})$ between two abstract states $e \overset{g}{\rightsquigarrow} e'$, where e = $\{t_1 \to l^{t_1}, ..., t_j \to l^{t_j}, ..., t_n \to l^{t_n}\}$ and e' = $\{t_1 \to l^{t_1}, ..., i_j \to l^{t'_j}, ..., t_n \to l^{t_n}\}$.

If the operation op contains the *pthread_create* statement for thread t_i, the transfer realtion addes a new thread t_{new} with the location $l^{\mathrm{i}_{\text{new}}} \in L$ to the abstract state e':

$$e' = e \setminus \{t_j \to l^{t_j}\} \cup \{t_j \to l^{t'_j}\} \cup \{t_{\text{new}} \to l^{\mathrm{t}_{\text{new}}}\}.$$

If the operation op contains the *pthread_join*(t_{join}) statement for thread t_j, the transfer realtion waits for a thread t_{join} to exit at program location $\mathrm{l}^{t_{\text{join}}}$:

$$e' = e \setminus \{i_{\text{join}} \to l^{\mathrm{t}_{\text{join}}}\} \setminus \{t_j \to l^{t_j}\} \cup \{t_j \to l^{t'_j}\}.$$

All other operations are not related to the thread management. The thread t_j simply moves from location l^{t_j} to location $l^{t'_j}$:

$$e' = e \setminus \{t_j \to l^{t_j}\} \cup \{t_j \to l^{t'_j}\}.$$

2.3.2 PredicateCPA

PredicateCPA $\mathbb{P}$ is the core component of CPAchecker for predicate-based analysis that was defined in [BDW17]. Its primary purpose is to construct SMT formulas representing all program paths leading to error location, which can later be checked with an SMT Solver for feasibility. The **PredicateCPA** $\mathbb{P} = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, merge_{\mathbb{P}}, stop_{\mathbb{P}}, prec_{\mathbb{P}})$ also conforms to the structure of configurable program analysis.

The **abstract domain** $D = (C, \mathcal{E}_{\mathbb{P}}, [\![\cdot]\!]_{\mathbb{P}})$ consists of the set C of concrete states, the semilattice $E_{\mathbb{P}}$ over abstract states, and the concretization function $[\![\cdot]\!]_{\mathbb{P}}$. The semilattice $\mathcal{E}_{\mathbb{P}} = (E_{\mathbb{P}}, \sqsubseteq_{\mathbb{P}})$ is a partial order set of abstract states $E_{\mathbb{P}}$. An abstract state $e \in E$ of PredicateCPA is a triple (ψ, l^{ψ}, ϕ) of an abstraction formula ψ, the abstraction location l^{ψ} (the program location of computed abstraction formula), and a path formula ϕ from the last abstraction state to the current abstract state. Both formulas are first-order formulas over predicates over the program variables represented as SMT formulas. Abstract states where abstraction is computed (path formula $\phi = true$) are denoted as abstraction states, otherwise intermediate states. The following implication $(\psi_1, l^{\psi_1}, \phi_1) \sqsubseteq_{\mathbb{P}} (\psi_2, l^{\psi_2}, \phi_2) = ((\psi_1 \wedge \phi_1) \implies (\psi_2 \wedge \phi_2))$ defines a partial order $\sqsubseteq_{\mathbb{P}}$, meaning an abstract state e_1 is less than or equal to another state e_2.

A **precision** $\pi \in \Pi$ of the PredicateCPA maps program locations to sets of predicates over the program variables. It allows using an appropriate set of predicates at any program location for abstraction computation. Additionally, some predicates may be used for all program locations or within a specific function.

If there exists a CFA edge $g = (l_i, op_i, l_j)$, the **transfer relation** $(\psi, l^{\psi}, \phi) \rightsquigarrow ((\psi, l^{\psi}, \phi'), \pi)$ computes an abstract successor $e' \in E$, which is an intermediate state. Comparing to the predecessor abstract state $e \in E$, the abstract successor's abstraction formula ψ and the abstraction computation location l^{ψ} stay the same. The transfer relation changes only the path formula of a successor by constructing the conjunction of the predecessor's path formula ϕ and the operation op_i of the current CFA edge: $\phi' = \phi \wedge op_i$

The **merge operator** $merge_{\mathbb{P}}$ combines only intermediate states that are located in the same block. It means that these abstract states have the same abstraction formula ψ and abstraction computation location l^{ψ}. Merge operator $merge_{\mathbb{P}}$ is defined as:

$$merge_{\mathbb{P}}((\psi_1, l^{\psi}{}_1, \phi_1), (\psi_2, l^{\psi}{}_2, \phi_2), \pi) = \begin{cases} (\psi_2, l^{\psi}{}_2, \phi_1 \vee \phi_2) & \text{if } (\psi_1 = \psi_2) \wedge (l^{\psi}{}_1 = l^{\psi}{}_2) \\ (\psi_2, l^{\psi}{}_2, \phi_2), & \text{otherwise} \end{cases} \tag{2.1}$$

The coverage checks for predicate-based analysis require solving SMT queries that influence the performance of program analysis. Thus, the **stop operator** *stop* was restricted to check coverage only for abstraction states and return false for intermediate states. The stop operator checks if there exists an abstraction state $(\psi', l^{\psi'}, \phi')$ in the set of abstract states R whose abstraction formula ϕ' is implied by the abstraction formula ϕ of the current abstraction state (ψ, l^{ψ}, ϕ).

$$stop_{\mathbb{P}}((\psi, l^{\psi}, \phi), R, \pi) = \begin{cases} \exists (\psi', l^{\psi'}, \phi') \in R : \phi' = true \wedge (\psi, l^{\psi}, \phi) \sqsubseteq_{\mathbb{P}} (\psi', l^{\psi'}, \phi') & \text{if } \phi = true \\ false & \text{otherwise} \end{cases} \tag{2.2}$$

The **precision-adjustment** operator *prec* is combined with a technique called adjustable-block encoding (ABE) [BKW10] that decides whether the abstraction should be computed at the current abstract state. If the operator *blk* returns *true* (current block ends), the operator *prec* converts an intermediate state into an abstraction state by computing the new abstraction. Otherwise, it returns the inputted abstract state with precision. In the first case, the precision-adjustment operator takes an intermediate state (ψ, l^{ψ}, ϕ) at program location l and computes the boolean predicate abstraction $(\psi \wedge \phi)^{\pi(l)}{}_{\mathbb{B}} = (\psi \wedge \phi) \wedge \bigwedge_{p_i \in \pi(l)} (v_i \Leftrightarrow p_i)$ for the abstraction formula ψ, path formula ϕ, and the set of predicates $\pi(l)$ from the precision π. We also assign a new variable v_i to each predicate $p_i \in \pi(l)$.

$$prec_{\mathbb{P}}((\psi, l^{\psi}, \phi), R, \pi) = \begin{cases} (((\psi \wedge \phi)^{\pi(l)}{}_{\mathbb{B}}, l, true), \pi) & \text{if } blk(\psi, l^{\psi}, \phi), l) \\ (\psi, l^{\psi}, \phi), \pi) & \text{otherwise} \end{cases} \tag{2.3}$$

2.3.3 CPA Algorithm

The CPA algorithm is a reachability algorithm that performs a simple state-space exploration. Its main purpose is to compute a set of all abstract states that can be reached from an initial state. The CPA algorithm's significant advantage is that it can be used with the composition of any CPAs. This flexibility allows for combining the CPA algorithm with different approaches like data-flow analysis and model checking. This paragraph describes the extended CPA+ algorithm 2.1 for configurable program analysis with the dynamic precision adjustment[BHT08]. The CPA Algorithm was extended to support the counterexample-guided abstraction refinement with a lazy abstraction and features of the PredicateCPA. These changes allow terminating the algorithm as soon as it reaches an abstract error state, to perform a refinement step and restart the algorithm from the same point. So we can expand further the set of abstract states without restarting from scratch.

The CPA+ algorithm takes as input a CPA D, sets *reached* and *waitlist* of abstract states with precision and function *abort*. The *reached* set contains all abstract states with precisions that have already been processed. All abstract states, whose successor states should be explored, are stored in the *waitlist* set. The algorithm keeps updating two sets by looping until either all abstract states have been fully processed (*waitlist* is empty) or function *abort* determines that the algorithm should terminate earlier. The abort(e) function typically checks if $e \in E$ is an abstract state at error location l_{err} and returns true to stop the algorithm. After the CPA+ algorithm aborts, it returns an updated version of both *reached* and *waitlist* sets.

```
Input: a CPA $D = (D, \Pi, \rightsquigarrow, merge, stop, prec)$,
  where $E$ denotes the set of elements of the semilattice of $D$,
  a set reached $\in E \times \Pi$ of reachable abstract states,
  a set waitlist $\in E \times \Pi$ of frontier abstract states,
  a function abort : $E \rightarrow B$ that defines whether the algorithm
      should abort early
Output: the updated sets reached and waitlist

while waitlist $\neq \emptyset$ do
  pop $(e,\pi)$ from waitlist
  if $(e,\pi) \notin$ reached then
   continue
  for all $e'$ with $e \rightsquigarrow (e',\pi)$ do
    $(\overline{e},\overline{\pi})$ := prec($e'$, $\pi$, reached)
    for all ($e''$, $\pi''$) $\in$ reached do
      $e_{\text{new}}$ := merge($\overline{e}$,$e''$,$\overline{\pi}$)
      if $e_{\text{new}} \neq e''$ then
        waitlist := waitlist $\cup$ {($e_{\text{new}}$,$\overline{\pi}$)}\{($e''$, $\pi''$)}
        reached := reached $\cup$ {($e_{\text{new}}$,$\overline{\pi}$)}\{($e''$, $\pi''$)}
    if not stop($\overline{e}$,{$e$ | ($e$, $\cdot$) $\in$ reached},$\overline{\pi}$) then
      waitlist := waitlist $\cup$ {($\overline{e}$,$\overline{\pi}$)}
      reached := reached $\cup$ {($\overline{e}$,$\overline{\pi}$)}
      if abort($\overline{e}$) then
        return (reached, waitlist)
return (reached, waitlist)
```

Listing 2.1: CPA+ algorithm

In each iteration, the algorithm takes an abstract state e with precision from the *waitlist* and computes all abstract successors e' according to the transfer relation $\rightsquigarrow$ of the CPA D. If the state exploration started from scratch, the *waitlist* and *reached* sets contain only an initial abstract state e_0 with precision Π_0. The algorithm adjusts the precision of each successor using the precision adjustment function *prec*. Next, all abstract successors e' are merged with each of the existing abstract state e'' in reached by using the given *merge* operator of CPA D. As a result, a new abstract state e_{new} is created that may differ from e''. If e_{new} contains additional information, such that the old information is strictly subsumed, then the old abstract state e'' with precision Π'' in the *reached* and *waitlist* sets is replaced by the abstract state e_{new} with precision Π_{new}. After the merge step, the algorithm checks using the *stop* operator of D if the current abstract state is already covered by the set *reached*. If it is the case, the exploration of this abstract state is stopped. Otherwise, the abstract state and its precision are added to the sets *reached* and *waitlist*. In case the *waitlist* is not empty, the algorithm continues to process another abstract state form waitlist in the next iteration. If all abstract states have been explored, the algorithm terminates and returns the reached with all processed states.

CPA Algorithm unrolls the program lazily into an abstract reachability graph (ARG). ARG is a directed acyclic graph representing all program paths that can be traversed during program execution, starting from the entry point. Its nodes are program abstract states and edges represent the transfer relation that leads from one abstract state to the next one. In order to construct the abstract reachability graph during program analysis CPAchecker uses the ARG CPA that stores the predecessor – successor relationship between abstract states. It allows then to reconstruct abstract paths from the ARG, which is a sequence $\langle e_0, ..., e_n \rangle$ of abstract states.

2.4 Example

The following example shows how CPAchecker performs the state-space exploration of the program (Listing 2.2). The program contains two global variables i and j, which are both initialized to 1. Then an additional thread is created that reads and modifies the values of global variables. In line 15, the main method checks the assignment of global variables. If either i or j is equal to 8, the program reaches the error state in line 16.

```
 1  int i=1, j=1;
 2  void *t1(void* arg)
 3  {
 4    i+=j;
 5    i+=j;
 6  }

 8  int main(int argc, char **argv)
 9  {
10    pthread_t id1;
11    pthread_create(&id1, ((void *)0), t1, ((void *)0));
12    j+=i;
13    j+=i;
14    pthread_join(id1, ((void *)0));
15    if (i == 8 || j == 8) {
16      ERROR: {reach_error();abort();}
17    }
18    return 0;
19  }
```

Listing 2.2: Program with concurrent threads

Firstly, CPAchecker parses the program and constructs the corresponding

CFA represented in Fig. 2.2, where each function is represented as a separate CFA. Every subsequent location is connected with the predecessor by an edge labeled with a program statement. The starting state l_0 is the initial location of the program. The corresponding error state is represented as an error location $l_{err} = l_8$.

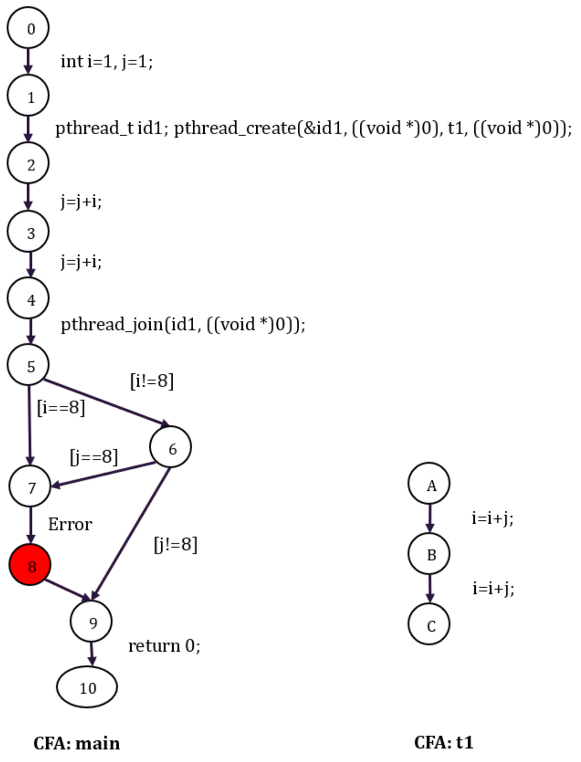

Figure 2.2: CFA for the functions of program 2.2

After constructing the program's CFA, CPAchecker explores the state-space of the program using the CPA Algorithm and ThreadingCPA. This example is simplified, and most program analyses require combining the ThreadingCPA with other CPAs. For example, in order to track predicates or variable assignments. The program analysis starts at the initial location l_0 of the main function and analyzes all possible thread interleavings. After the control flow reaches the statement *pthread_create* in location l_2 of the main function, ThreadingCPA tracks an additional thread with its program location. The diamond-like structure in the ARG results from merging two interleaving threads when reaching the same program location via different execution paths. When the control flow executes the statement *pthread_join* in location l_5, the program location of the existing thread is removed from the abstract state.

The result of the program state-space exploration is the ARG in Fig. that represents all program paths traversed during the program execution. Each abstract state is labeled with the indices of the program locations of all currently active threads.

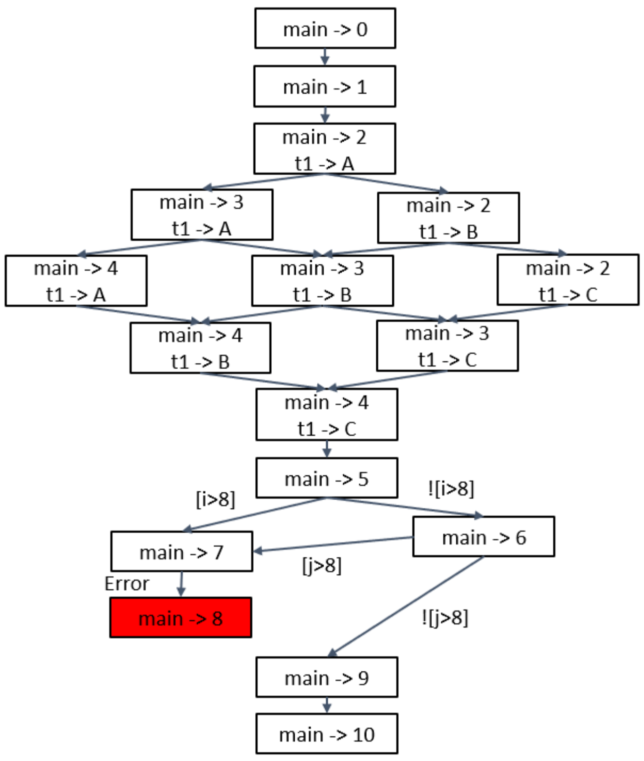

Figure 2.3: ARG of the interleaved threads of the program 2.2

2.5 SMT-Based Approaches for Software Verification of concurrent programs

Bounded Model Checking with adjustable-block encoding, Predicate Abstraction with counterexample-guided abstraction refinement, and lazy abstraction are the techniques that scaled software verification from simple programs to complex industrial software[BDW17]. This section provides an overview of these widely used predicate-based software verification approaches that are based on SMT solving as the back-end technology. The core component for predicate-based analysis is the PredicateCPA that enables the expression of various techniques within one framework. The implementation of these verification approaches within a CPAchecker requires combining an appropriately configured instance of the PredicateCPA with other relevant CPAs using CompositeCPA. In addition, we use an underlying CPA+ algorithm, which is wrapped inside each analysis, to explore the state-space of the program and construct corresponding ARG.

2.5.1 Bounded Model Checking

The main idea of Bounded Model Checking (BMC) is to check whether a property violation can occur in k steps during the program execution. This approach is able to verify a large state-space of different program types in a short time without expensive abstraction computations, coverage checks, and refinements. Instead, BMC performs bounded unrolling of all program paths in an SMT formula that is checked later with SMT Solver for feasibility. If the satisfiable SMT formula encodes a program path leading from the program entry to an error location, the property violation of specification is detected. Otherwise, we can increase the bound k until either any feasible program path to the error location is finally found, or all possible violations have been excluded.

To verify concurrent programs, BMC takes as input the CompositeCPA that consists of ThreadingCPA, PredicateCPA, and the LoopBoundCPA. Each abstract state is a tuple of threading abstract state (containing current active threads with locations), predicate abstract state (tuple of an abstraction formula, an abstraction location, and a path formula), and loop-bound abstract state (maps loop heads to loop counters). The LoopBoundCPA is used to track a loop counter in its abstract states for every program loop. By determining how many times the loop body has already been traversed on the current program path, LoopBoundCPA prevents exploring any loop after k iterations. The transfer relation of the LoopBoundCPA does not compute a successor for abstract states, which contains a loop counter equal to loop bound k.

Bounded Model Checking starts with the program state-space exploration performed by the CPA+ algorithm. In order to avoid abstraction computation at program block ends, the ABE block size is set to infinite that enables whole program encoding. Thus, a path formula of each abstract state in PredicateCPA encodes exactly the path from the initial location to the current abstract state. After the CPA+ algorithm has computed the set of all reachable abstract states, BMC constructs a single SMT formula containing a disjunction of all path formulas leading to error locations. If at least one path formula is satisfiable, a feasible path to the error location is found that violates the specification.

```
Input: the initial value $k_{\mathrm{INIT}} \geq 1$ for the bound $k$,
  an upper limit $k_{\mathrm{MAX}}$ for the bound $k$,
  a function $inc : \mathbb{N} \to \mathbb{N}$ with $\forall n \in \mathbb{N} : inc(n) > n$ for increasing the
      bound k,
  a CompositeCPA $D$ with the ThreadingCPA $T$, the PredicateCPA $P$,
      and the LoopBoundCPA $LB$ as components,
  for which $E$ denotes the set of composite abstract states and $\Pi$
      the set of precisions.
Output: $false$ if $l_{\mathrm{err}}$ is reachable, $true$ otherwise
Variables: the current loop bound $k \in N$,
  two abstract states $e_{\mathrm{INIT}} \in E$ and a precision $\pi_{\mathrm{INIT}} \in \Pi$,
  two sets reached and waitlist of elements of $E \times \Pi$, and
  a function $abort : E \to true, false$

$k := k_{\mathrm{INIT}}$
$e_{\mathrm{INIT}} := (l_{\mathrm{INIT}}, (true, l_{\mathrm{INIT}}, true))$
$abort^{\mathrm{NEVER}} := \cdot \to false$
while $k \leq k_{\mathrm{MAX}}$ do
  $\pi_{\mathrm{INIT}} := \{(\emptyset, \{\cdot \to \emptyset\}, k)\}$
  reached := waitlist := $\{(e_{\mathrm{INIT}}, \pi_{\mathrm{INIT}})\}$
  (reached, waitlist) := CPA++($D$, reached, waitlist, $abort^{\mathrm{NEVER}}$)
  $base\_case := \bigvee \{\varphi \mid ((l_{\mathrm{err}}, (\cdot, \cdot, \varphi), \cdot), \cdot) \in reached\}$
  if sat($base\_case$) then
    return false

  k := inc(k)
retrun unknown;
```

Listing 2.3: Bounded Model Checking

2.5.2 Lazy Predicate Abstraction

Abstraction is the unbounded model checking technique that attempts to compute an overapproximating abstract model of the program in order to avoid the state-space explosion. This approach simplifies a program by excluding the program semantics that is irrelevant to prove or disprove its safety. On the one hand, the abstraction should be coarse enough to keep the state space small. On the other hand, too coarse abstraction may cause the incorrect result of the program analysis. An overapproximation usually does not satisfy the same properties as the original program. Thus the refinement is performed to adjust the precision of analysis. The computation of the state space's overapproximation is usually combined with the counterexample-guided abstraction refinement.

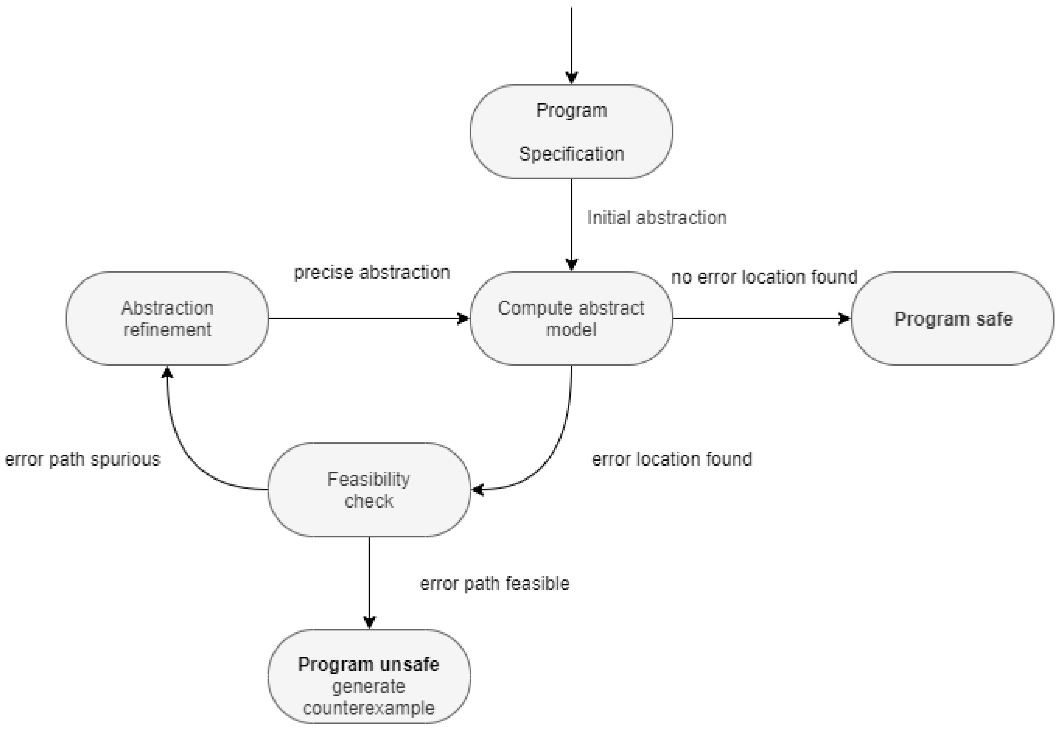

Figure 2.4: The workflow of counterexample-guided abstraction refinement (CEGAR).

Counterexample-guided abstraction refinement (CEGAR) is an approach that uses counterexamples to refine an abstract model iteratively. A counterexample is a program path that leads to an error location and proves property violation. Starting with the coarse abstraction (empty set of predicates), CEGAR uses a CPA+ algorithm to explore the state-space of the program and create the abstract model. If the algorithm finds an abstract state at error location l_{err}, further the state-space exploration is paused, and the refinement is started. The operator $refine$ reconstructs the error path using the predecessor – successor relationship between abstract states stored in ARG CPA and checks the feasibility of the path by solving an SMT formula. If the error path is feasible, the CEGAR terminates and reports that the program is unsafe, containing a counterexample. Otherwise, the error path is infeasible, which indicates that the violation is detected due to a too coarse abstract model. Then the infeasible error path is used to refine the abstract model. In addition, we remove the subgraph of the ARG with abstract states at locations for which new predicates were computed. Due to new precision, more predicates will be used in the abstraction computation that would lead to more accurate program analysis. After the refinement step, the analysis is restarted with adjusted sets *reached* and *waitlist* and more strong precision to eliminate the infeasible error path in further state-space exploration. CEGAR repeats these steps iteratively until either a counterexample is found or the abstract model is proven to be safe. [BL12]

CEGAR usually is combined with the concept of lazy abstraction that refines only abstract states along infeasible error paths. Thus, the precision is increased selectively in specific parts of ARG to prevent further discovering of infeasible error paths, making the state-space exploration much more efficient. In addition, CEGAR uses the coverage checks that reduce the number of abstract states by determining whether any other abstract state covers a new abstract state. The abstraction formula of an abstract state is computed, and the coverage check is performed only if the CPA+ algorithm reaches the block end. The abstraction formula (represented as SMT formula) is boolean conjunction of predicates from precision π that overapproximates concrete program states. [BDW17]

```
Input: a CompositeCPA $D$ that is composed of the ThreadingCPA $T$,
   the ARG CPA $A$, and the PredicateCPA $P$, for which $E$ denotes the
    set of composite abstract states and $\pi$ the set of precisions,
    with additional operator $refine$, and an initial abstract state
   $e_{\mathrm{INIT}} = (l_{\mathrm{INIT}}, ...) \in E$ with initial precision $\pi_{\mathrm{INIT}} \in \Pi$
Output: $false$ if $l_{\mathrm{err}}$ is reachable, $true$ otherwise
Variables: two sets reached and waitlist of elements of $E \times \Pi$ and a
    function $abort : E \to true, false$

reached := {($e_{\mathrm{INIT}}$, $\pi_{\mathrm{INIT}}$)}
waitlist := {($e_{\mathrm{INIT}}$, $\pi_{\mathrm{INIT}}$)}
loop
  (reached, waitlist) := CPA++($D$, reached, waitlist, abort)
  if $\exists((l_{\mathrm{err}}, ...), .) \in$ reached then
    (reached, waitlist) := refine(reached, waitlist)
    if $\exists((l_{\mathrm{err}}, ...), .) \in$ reached then
      return false //refine has detected a feasible error path.
    else
      return true
```

Listing 2.4: Counterexample-guided abstraction refinement (CEGAR)

Chapter 3

Implementation

The following section describes the implemented extensions of existing components that allow us to verify multithreaded programs using Bounded Model Checking and Predicate Abstraction approaches. We also implement some optimizations to increase program analysis efficiency and use the full potential of our framework.

3.1 PredicateCPA

Firstly, we extend the core component for predicate-based analyses PredicateCPA to be compatible with ThreadingCPA, which is able to track multiple program locations simultaneously. As outlined in the previous section, an abstract state $e \in E$ of the PredicateCPA is a triple (ψ, l, ϕ) of an abstraction formula ψ, the abstraction location l, and a path formula ϕ. We extend an abstract state of PredicateCPA to a triple (ψ, L, ϕ), which now contains a set L of program locations instead of a single location l. These program locations determine where the abstraction ψ was computed.

We should also extend the precision-adjustment operator *prec* of PredicateCPA, which either transforms an intermediate state into an abstraction state by computing predicate abstraction or returns the given intermediate state and precision. Whether an abstraction has to be computed at the current abstract state depends on the block-adjustment operator *blk*, which decision is based on information about the current program location and abstract state. The abstraction is necessary for detecting whether an abstract state is reachable by performing the satisfiability check. In our case, the block operator should be able to consider multiple locations per abstract state. Thus, we use the existing block operator and compute abstractions if any of the available locations reaches a block-end. For example, the block operator blk^{lf} behaves similarly to large-block encoding (LBE) and returns true at loop-heads, function calls/returns, and at the error location l_{err}. It has been shown that in practice, boolean abstraction is too expensive for single-block encoding (SBE). Suppose we apply the block operator blk^{lf} to the program 2.2. In that case, a precision-adjustment operator will compute the abstraction at states with locations $t_1 \to A$ (t1 function call), $t_1 \to C$ (t1 function return), and $main \to 8$ (error location). The abstract states with a computed abstraction are presented at the following ARG (Figure 3.1) as the highlighted nodes.

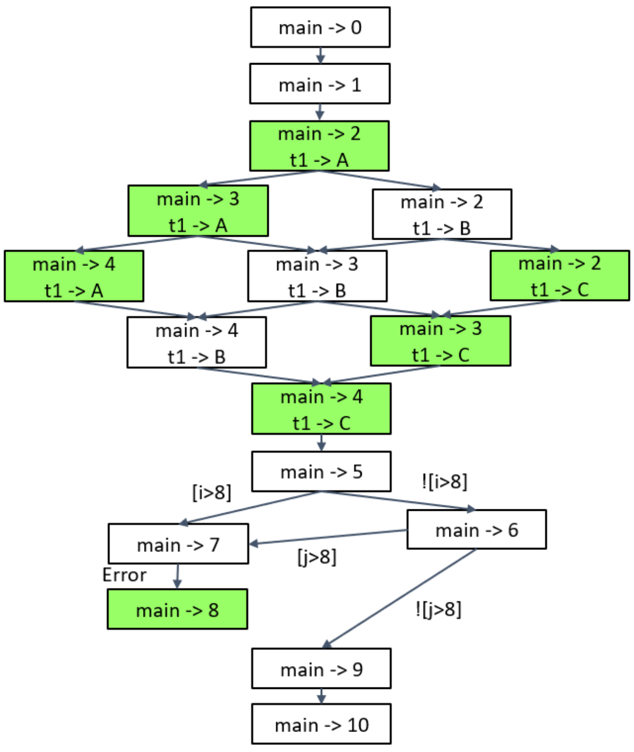

Figure 3.1: ARG of the interleaved threads of the program 2.2 with highlighted abstraction states

Let L be a set of all program locations that are tracked by the ThreadingCPA for an abstract state. To create an abstraction state from an intermediate state (ψ, L, ϕ) at program locations L, the precision-adjustment operator should compute a predicate abstraction $(\psi \wedge \phi)^{\pi(L)}{}_{\mathbb{B}}$ with respect to multiple program locations from set L. Newly computed predicate abstraction is a conjunction of the abstraction formula ψ, path formula ϕ, and the conjunction of all predicates p_i from the precision π for all program locations from set L.

$$(\psi \wedge \phi)^{\pi(L)}{}_{\mathbb{B}} = (\psi \wedge \phi) \wedge \bigwedge_{l \in L} (\bigwedge_{p_i \in \pi(l)} (v_i \Leftrightarrow p_i)) \tag{3.1}$$

Therefore, we redefine precision-adjustment operator *prec* of Predicate-CPA, which is now able to handle multiple program locations from set L while computing new abstraction:

$$prec_{\mathbb{P}}((\psi, L^{\psi}, \phi), R, \pi) = \begin{cases} (((\psi \wedge \phi)^{\pi(L)}{}_{\mathbb{B}}, L, true), \pi) & \text{if } blk(\psi, L^{\psi}, \phi), L) \\ (\psi, L^{\psi}, \phi), \pi) & \text{otherwise} \end{cases} \tag{3.2}$$

3.2 Refinement

Counterexample-guided abstraction refinement (CEGAR) is an approach that iteratively finds an analysis precision that should be precise and coarse enough to provide an efficient and correct analysis. The refinement process of the analysis precision is started when the CEGAR finds a spurious program path, which leads to an abstract state at an error location. New analysis precision, which maps program locations to sets of predicates, ensures that the same error path will not be encountered after the analysis is restarted after refinement. Since we want to extend the CEGAR approach to support the verification of concurrent programs, the underlying reachability analysis should be stopped as soon as any thread reaches an error location.

Predicate refinement is a common refinement strategy for lazy Predicate Abstraction that computes a Craig interpolant for each location on the path and extends the previous precision with new predicates extracted from interpolants.[BDW17] The refinement operator *refine* uses an infeasible program path $\langle e_0, ..., e_n \rangle$ with abstraction states $e_i \in E$ at program locations $\langle l_0, ..., l_m \rangle$ to extract a sequence $\langle \rho_0, ..., \rho_n \rangle$ of sets of predicates. The extraction of the atoms from the interpolants as predicates is performed concerning multiple program location pro abstract state. Each predicate is added to the precision for the corresponding program location.

$$\pi_{\text{new}}(l) = \begin{cases} \bigcup_{i=0}^{n} \rho_i & l \in \langle l_0, ..., l_m \rangle \\ \emptyset & \text{otherwise} \end{cases} \tag{3.3}$$

Finally, we combine the newly computed precision π_{new} with the already existing one π_{old} for each program location in the spurious error path.

$$\forall l \in \langle l_0, ..., l_m \rangle : \pi(l) = \pi_{\text{new}}(l) \cup \pi_{\text{old}}(l) \tag{3.4}$$

After the refinement process, the CEGAR continues building the abstract model with adjusted sets *reached* and *waitlist* and more strong precision.

3.3 Simplification of SMT formula

Bounded Model Checking and Predicate Abstraction are techniques that rely on SMT solving. A satisfiability modulo theories (SMT) problem is a generalization of a boolean satisfiability (SAT) problem in which sets of variables are replaced by predicates from underlying theories. Moreover, the SAT is the first problem proven to be NP-complete, so no algorithm efficiently solves this problem. In order to determine whether the program state is reachable, the corresponding SMT formula should be checked for satisfiability. This problem requires determining whether there exist such variable assignments that satisfy a given SMT formula. Analyzing all possible paths of a single program demands a large amount of computation time for solving corresponding SMT formulas. It can make the analysis not to be able to verify the program due to time or resource constraints.

This section introduces an algorithm for simplifying boolean formulas (Listing 3.1) that we apply to the SMT formula constructed by Bounded Model Checking. The final SMT formula is represented as a disjunction of all path formulas leading to error locations reachable during state-space exploration. If some path formulas encode the same part of the program, they contain some common predicates that make the final SAT check more complicated and time-consuming. Our algorithm applies standard algebraic reduction rules such as the distributive law of $\wedge$ over $\vee$ to a given boolean formula. Its basic idea is to split the formula containing a disjunction into a set of boolean subformulas. Then it iterates through all subformulas, which are conjunctions of predicates, and factors out all common predicates. Furthermore, each subformula may contain additional disjunctions of predicates that also should be simplified recursively. Finally, the simplified boolean formula is equivalent to the original one but contains fewer predicates that enables to reduce the time required for SMT solver to check the given formula. In section 4, we evaluate the Bounded Model Checking combined with the simplification technique for boolean formulas on the set of benchmarks.

```
Function: simplifyBooleanFormulaRecursively
Input: boolean formula to be simplified,
  a function toDisjunction() that splits a given boolean formula containing a disjunction into a set of boolean subformulas,
  a function toConjunction() that splits a given boolean formula containing a conjunction into a set of boolean subformulas,
  a function isSinglePredicate() that returns true if a given boolean formula is a single predicate.
Output: simplified formula

cachedBooleanFormulas = {}
listOfOperandsSets := []
disjunctionSet := toDisjunction(formula)
for each subformula ∈ disjunctionSet do
  conjunctionSet := toConjunction(subformula)
  operandsOfSubformula := {}
  listOfOperandsSets := listOfOperandsSets ∪ {operandsOfSubformula}
  for each subformulaInConjunctionSet ∈ conjunctionSet do
    if isSinglePredicate(subformulaInConjunctionSet)
      operandsOfSubformula := operandsOfSubformula ∪ {subformulaInConjunctionSet}
    else
      operandsOfSubformula := operandsOfSubformula ∪ {simplifyBooleanFormulaRecursively(subformulaInConjunctionSet)}

mutualOperandsSet := listOfOperandsSets[0]
for each operandsOfSubformula ∈ listOfOperandsSets \ listOfOperandsSets[0] do
  mutualOperandsSet := mutualOperandsSet ∩ operandsOfSubformula

transformedSubformulas := []
for each operandsSet ∈ listOfOperandsSets do
  operandsSet := operandsSet \ mutualOperandsSet
  transformedSubformulas := transformedSubformulas ∪ { $\bigwedge_{op \in operandsSet} op$ }

return $(\bigwedge_{op \in mutualOperandsSet} op) \bigwedge (\bigvee_{f \in transformedSubformulas} f)$
```

Listing 3.1: Algorithm for simplification of SMT formula

The algorithm takes as input a boolean formula that has to be simplified, functions *toDisjunction*() and *toConjunction*() that split a given boolean formula into a set of boolean subformulas, and a function *isSinglePredicate*() that returns true if a given boolean formula is a single predicate and not a combination of predicates. The first step is to split a given boolean formula into a set of subformulas using the function *toDisjunction*(). Also, we initialize an empty list *listOfOperandsSets* where each element will be a set of all operands in a single subformula needed for finding common predicates. Then we split each subformula using a function *toConjunction*() into a set containing all operands of current conjunction. In addition, we create an empty set *operandsOfSubformula* for operands of current subformula and add it to *listOfOperandsSets*. Then we iterate through all operands of current conjunction and check whether it is a single predicate. If it is a case, we add the predicate to the set *operandsOfSubformula*. Otherwise, the operand is a boolean formula containing a disjunction that can be simplified. Thus, we apply our algorithm recursively to the operand and add it to the set *operandsOfSubformula*. Additionally, we cache all processed subformulas to avoid the redundant simplification of equal subformulas. Then we build an intersection of all sets in a *listOfOperandsSets* containing operands of each subformula and store all common operands in a set *mutualOperandsSet*. The next step is to remove all mutual operands from each *operandsSet* of a *listOfOperandsSets* and store the conjunctions of operands from each *operandsSet* in a set *transformedSubformulas*. Finally, we construct the simplified formula by putting out of disjunction of *transformedSubformulas* a conjunction of all operands from *mutualOperandsSet*.

The following example displays a simplification of the boolean formula that represents a simple program. Let A, B, C, D, E be predicates over program variables and the boolean formula:

$$(A \land B \land C) \lor (A \land B \land D \land E) \tag{3.5}$$

Firstly, the algorithm splits a disjunction and iterates through subformulas $(A \land B \land C)$ and $(A \land B \land D \land E)$ to split each conjunction and store all predicates in *listOfOperandsSets*.

$$listOfOperandsSets = \{\{A, B, C\}, \{A, B, D, E\}\} \tag{3.6}$$

Then the algorithm computes the set of common predicates *mutualOperandsSet* and remove them from each set stored in *listOfOperandsSets*.

$$mutualOperandsSet = \{A, B\} \tag{3.7}$$
$$listOfOperandsSets = \{\{C\}, \{D, E\}\} \tag{3.8}$$

The remaining operands in *listOfOperandsSets* are used to construct a set *transformedSubformulas*, which contains subformulas without mutual operands.

$$transformedSubformulas = \{C, D \land E\} \tag{3.9}$$

The final step is to build a conjunction of operands from $mutualOperandsSet$ and disjunction of $transformedSubformulas$. Thus, we obtain the simplified boolean formula:

$$(A \land B) \land (C \lor (D \land E)) \tag{3.10}$$

Chapter 4

Evaluation

In this section, we estimate the capabilities of Bounded Model Checking and Predicate Abstraction on a large set of benchmarks and compare them with other existing verification approaches that are implemented in the same framework. This section concludes with a discussion of the obtained results and lessons learned based on our implementation.

4.1 Benchmarks

To evaluate the effectiveness and performance of our verification approaches, we use the collection of verification tasks from the sv-benchmarks repository[1] in revision f86649d that were developed for the International Competition on Software Verification[2]. From the wide variety of available benchmark tasks that intend to test the potential of modern verification software, we focus only on 1082 tasks taken from the category of concurrent programs. Each program from the benchmark set is a multithread C program where the safety property is to be verified, meaning reaching a specific program location is considered as property violation. For example, some tasks contain concurrent access to shared variables, leading to data races and inappropriate program behavior.

[1]https://github.com/sosy-lab/sv-benchmarks/

[2]https://sv-comp.sosy-lab.org/2021/

4.2 Setup

The evaluation is performed on a computer cluster where each machine is equipped with a single Intel Xeon E3-1230 v5 8-core CPUs with 3.40 GHz, 33 GB of RAM, and runs Ubuntu (Linux 5.4.0-52) as an operating system. Resource usage is limited by the benchmarking framework BenchExec[BLW15], which enables reliable benchmarking and resource measurement for the comparative evaluation of tools and algorithms. Each verification task is executed on two CPU cores and is limited to 15 min of CPU run time and 15 GB of memory usage. For our experiments, we use the CPAchecker in version 1.9.2 in revision 35617 and the SMTInterpol[CJA12] as a standard SMT solver, which can be replaced by MathSAT5[CGSS13] to support the theories of bit-vectors and floats.

4.3 Configuration

The software-verification framework CPAchecker provides a wide variety of configurations and options for program analysis. It allows us to examine the actual performance of different verification approaches rather than comparing several tools that could influence the fair results due to differences in the implementation unrelated to the actual algorithms. To evaluate our implementations on the broad set of multithreaded programs, we created two configurations named `-bmc-concurrency` and `-predicateAnalysis-concurrency`. Furthermore, we run two additional configurations `-bddAnalysis-concurrency` and `-valueAnalysis-concurrency` to compare these techniques with Bounded Model Checking and Predicate Abstraction.

For the underlying reachability algorithm, we configured the CPAchecker to use the breadth-first traversing order (BFS), which turned out to be a more efficient strategy than depth-first order (DFS) for multithreaded programs. For the ThreadingCPA, we use the option that enables cloning functions by copying function names and inserting an index if the same function is called in different threads. It ensures all function-local variables to be unique for different threads in the later program analysis.

For Bounded Model Checking, we set a max loop bound to ten iterations for Loop-BoundCPA that prevents exploring any loop infinitely. We combinate the BMC approach with an explicit value analysis using ValueCPA that explicitly tracks the current value for each program variable. It allows us to prevent the exploration of unreachable program paths and increase the performance of the Bounded Model Checking. Besides, to evaluate the implemented optimization described in section 3.3, we run the second instance of the BMC algorithm with an additional option that simplifies the final SMT formula.

For Predicate Abstraction, an optimal ABE block size must to be set to perform the Predicate Abstraction computation and coverage checks at appropriate locations. Thus, based on the benchmark results, we choose the block operator *blk*^{l} as the most efficient approach that forces the abstraction computation only at loop heads. Also, we run an additional configuration of Predicate Abstraction combined with an explicit-value analysis to evaluate the performance of this approach applied to concurrent programs.

4.4 Results

The table 4.1 summarizes the experimental result of benchmark execution, displaying the number of both correctly and wrongly solved benchmark tasks, as well as the overall CPU time spent performing program analysis for each verification approach. Additionally, the table shows the number of tasks that terminated inconclusively because of the occurred error, memory overflow, timelimit, or reached loop bound. The uncertain results caused by an error occur if the specific program analysis encounters some unsupported operations or data structures.

The bdd and value analysis demonstrate the best results in reporting the most number of correct results and the lowest total CPU time. In contrast to the SMT-based verification approaches, they can eliminate the infeasible paths on the fly-way during the program analysis that gives them a significant performance. The explicit value analysis solved 946 out of the 1082 verification tasks correctly, making this technique the most successful among selected approaches for concurrent programs. The drawback of the value analysis is that it reports the most incorrect 'false' results due to its over-approximating nature[Löw17], alerting a specification violation for 57 correct verification tasks. According to the benchmark results, the bdd analysis is the second-best verification approach, which managed to solve 944 tasks. However, it has the most number of errors that happened during program analysis due to unsupported arrays and thread assignments.

The Bounded Model Checking also showed great benchmark results, solving 702 verification tasks and reporting the lowest amount of incorrect decisions. Therefore, BMC confirms its reputation as a straightforward and reliable technique for finding program bugs. In parallel, we execute the value analysis as an optimization for BMC, preventing unrolling unreachable program paths, reducing the program's state-space and time for its

Algorithm	**BMC**	**BMC with formula simplification**	**Predicate Abstraction**	**Predicate Abstraction with value analysis**	**BDD**	**Value analysis**
correct results	702	687	204	896	944	946
correct 'true'	152	152	168	178	167	162
correct 'false'	550	535	36	718	777	784
incorrect results	7	7	9	15	21	57
incorrect 'true'	1	1	2	2	0	0
incorrect 'false'	6	6	7	13	21	57
error	84	84	88	86	96	38
out of memory	165	201	1	2	0	9
timeouts	87	66	780	82	21	32
loop bound reached	37	37	0	0	0	0
total CPU time (s)	258000	266000	717000	224000	76600	84100

Table 4.1: Benchmark results for 1082 verification tasks from the category of concurrent programs.

exploration. Otherwise, the plain BMC algorithm would blindly discover all program paths that would make this approach inefficient because of the state-space explosion.

The main weakness of the BMC is the amount of time required for the final SAT check to prove whether the SMT-formula is feasible. The set of encountered program's paths and its length directly affects the size and complexity of the final SMT-formula, increasing the computation time of an SMT-solver. The implemented algorithm for boolean formula simplification intends to shorten the constructed SMT-formula by factoring out all common predicates. The BMC with formula simplification reported the same amount of incorrect results as the standard BMC configuration that indicates the correctness of our algorithm. The efficiency of the formula simplification we can see on `pthread-wmm/mix012_tso.opt.c` program, which contains various shared boolean variables and complex conditions checks. For this verification task, the final SAT check took only 17 seconds instead of 616 seconds after formula simplification that demonstrates the strengths of our approach. However, in some cases, the formula simplification may also increase the time for the final SAT check. In general, the Bounded Model Checking with formula simplification managed to solve 15 tasks less than the standard one due to the memory overflow. Based on benchmark results, the implementation of BMC turned out to be the most memory-consuming that caused the failure of 165 verification tasks for BMC and 201 for BMC with formula simplification. Nevertheless, the formula simplification algorithm can be further optimized to be more memory efficient, improving the general performance of the Bounded Model Checking approach.

From the resulting table, we can see that the Predicate Abstraction could not solve 780 verification tasks within the given time limit, providing only 204 correct decisions and showing the worst results among all approaches. The prime reason why the Predicate Abstraction fails to provide a verdict for a majority of verification tasks is a state-space explosion. This approach spends much time exploring all reachable program states, computing expensive abstractions, and performing refinements. Due to all possible thread interleavings, concurrent programs have a high branching rate, meaning there can exist a wide variety of different paths leading to an abstract error state. For such a verification task, precision must be computed for every error path found during the analysis. As a result, the process of finding a strong precision for program analysis is usually too expensive, and the verification runs out of time. During the verification of concurrent programs, Predicate Abstraction reaches a timeout after hundreds of refinements that do not produce a suitable precision. However, Predicate Abstraction combined with the value analysis is much more effective and outperforms even Bounded Model Checking. The additional value analysis helps to keep the discovered state-space narrow, reducing the number of required refinements and abstraction computations.

The following quantile plot (Figure 4.1) demonstrates the number of successfully solved tasks by a specific verification technique in the given amount of time. A data point (x, y) of a graph indicates that an individual approach was able to solve x verification tasks within y seconds of a CPU run time, showing the performance of each technique. The plot demonstrates only the data points for correct verification results, meaning the x-value corresponds to the number of correctly solved tasks. The monotonical growth of the graph is directly related to the performance of the verification technique.

The slower the corresponding graph grows, stretching to the right, the more efficient the verification technique is with the corresponding configuration. Moreover, a graph's slope may indicate the ability of the technique to scale for more complex tasks.

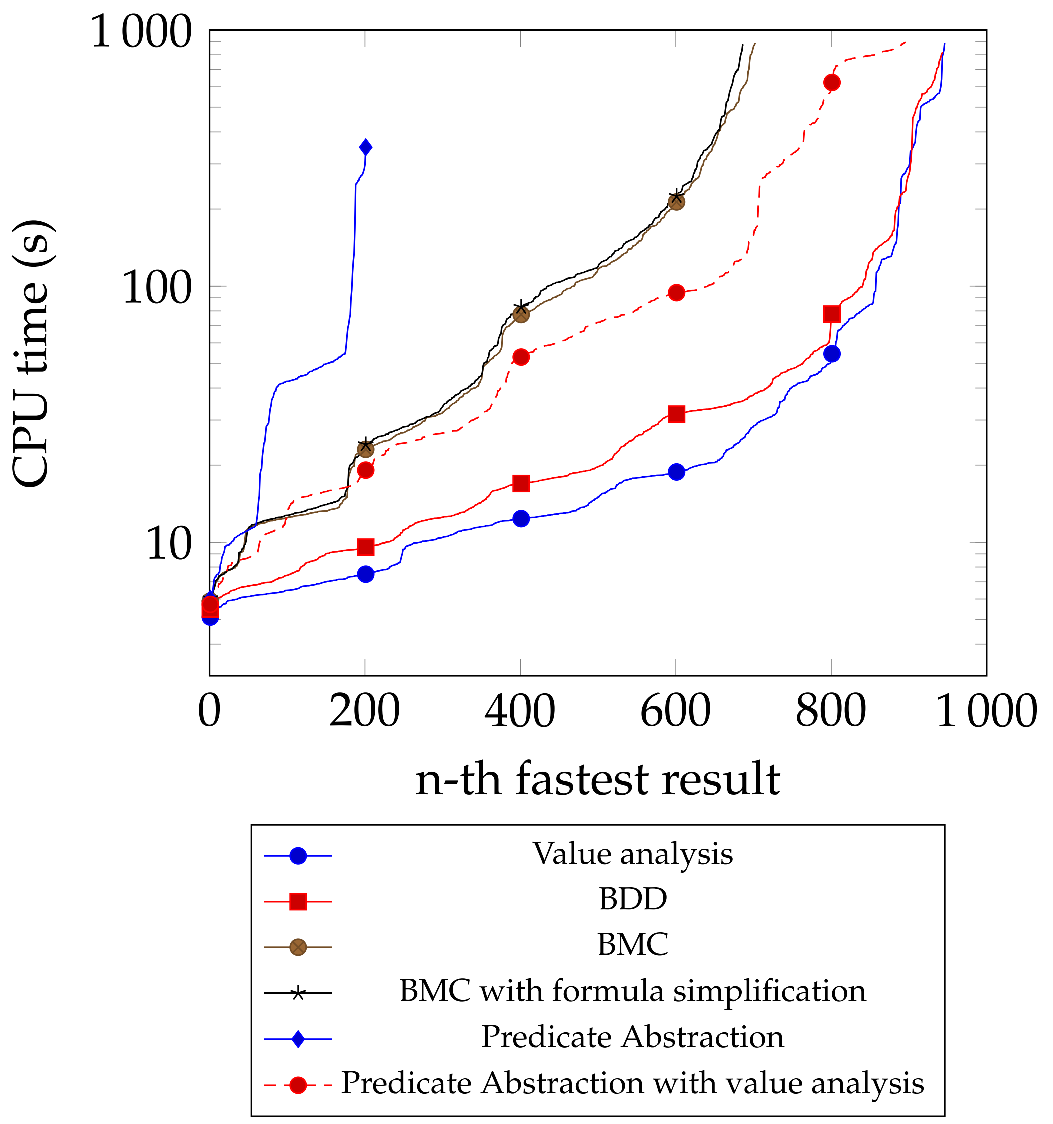

Figure 4.1: Quantile plot for all correct proofs of various verification approaches applied to concurrent programs.

Chapter 5

Related Work

The efficient verification and analysis of concurrent programs is challenging and offers a wide range of exciting research topics. Our SMT-based verification approaches explore all possible interleavings of different thread executions on-the-fly during the program analysis. It is possible due to the underlying core component ThreadingCPA that is independent of the applied analysis. Thus, we can combine the state-space exploration of multithreaded programs with various abstract domains for a specific program analysis type[BF16].

An alternative approach for Bounded Model Checking of multithreaded programs uses a specific sequentialization technique to transform the program on the sourcecode level before starting the analysis. Sequentialization extends SAT-based BMC to handle concurrent programs by reconstructing the initial program into a sequential one that preserves all the feasible execution paths. The Lazy-CSeq[ITF$^+$14] tool implements the sequentialization of the concurrent programs, which can be later analyzed by the C bounded model-checker CBMC[1] using the MiniSat[ES03] propositional solver as the default decision procedure. Technically, this technique decomposes the set of execution traces of the concurrent program into symbolic subsets, separately explored by multiple instances of the decision procedure running simultaneously. Each constructed path formula can be handled by a separate solver, terminating the whole analysis once one of them detects a satisfiable error path. Thus, each BMC instance for the non-parallel analysis can operate with different partitions of the program's state-space independently, using the computational capability of modern multi-core hardware and clusters[IT20].

[1]https://www.cprover.org/cbmc/

Chapter 6

Conclusion

This thesis aimed to present the implemented changes and optimizations in the software verification framework CPAchecker that allow us to verify concurrent programs using Bounded Model Checking and Predicate Abstraction approaches. Firstly, we studied the fundamental concepts of configurable program analysis that enable flexible and customizable program verification, combining two primary techniques such as program analysis and model checking. This architectural principle splits different program domains into independent components that can be adjusted and efficiently used for our verification approaches. Then we discussed the main components of CPAchecker's architecture and algorithms for Bounded Model Checking and Predicate Abstraction.

Secondly, we described the implementations in the core component for predicate-based analyses, as well as in the refinement strategy for Predicate Abstraction, required for verification of concurrent programs. These components are used to configure the underlying reachability analysis that explores the program's state-space analyzing all possible thread interleavings. We also implemented an additional optimization for Bounded Model Checking that factors out all common predicates in the final SMT formula. It showed a significant improvement in individual verification tasks, reducing the CPU time required for verification several times.

Finally, we presented the evaluation of our verification techniques and implemented optimizations on the broad set of concurrent verification tasks, comparing them with BDD and explicit-value analysis. Based on the benchmark results, SMT-based verification approaches suffer from the state-space explosion problem during the exploration of all reachable program states. Thus, to avoid this problem and improve the scalability of Bounded Model Checking and Predicate Abstraction, we presented a combination of our verification techniques with an explicit value-analysis. The value domain tracks each program variable's current value, eliminating the infeasible paths on the fly-way during the program analysis. This combination significantly reduces the number of SMT formulas representing corresponding program paths that have to be checked for feasibility.

6.1 Future work

There are various potential topics for future research, improving the performance and scalability of our SMT-based verification approaches. For instance, CPAchecker implements an adjustable-block encoding (ABE)[BKW10] that provides the flexibility to compute the abstraction after some configurable number of operations. For Predicate Abstraction, we use the block operator *blk*[1] to force the abstraction computation and coverage checks if any of the available locations reach a loop head. Based on the ARG at Fig. 3.1, we see that this strategy computes the abstraction for each abstract state along one path, where one of the available locations reaches the block end. The computation of the predicate abstractions after every single program operation is expensive. Thus, the one possible topic could be to determine the most efficient strategy of the block operator, as well as the block size for the verification of concurrent programs.

For Predicate Abstraction to achieve the successful and scalable verification of complex programs, the interpolation procedure could be improved that extracts interpolants from infeasible counterexamples. Generated interpolants for infeasible error paths are used to refine the precision of the state-space exploration algorithm.

The way of encoding program semantics into formulas and choice of the SMT solver can have a significant influence on the performance of the SMT-based analysis[Wen17]. The algorithm for simplification of SMT-formula could be further optimized to convert the formula into a specific form, which is easier to check for a particular SMT-solver. Based on the Bounded Model Checking benchmark results, the formula simplification can significantly reduce the CPU time required for the final SAT check. An alternative approach could be to simplify the SMT formula for each merge operation in PredicateCPA. In this case, we do not need to apply our algorithm recursive, which worst-case time complexity would be quadratic instead of exponential. However, it can be inefficient for complex programs with heavy branching because the simplification would be performed for every single merge operation. Moreover, the formula simplification could be used not only for Bounded Model Checking but also for Predicate Abstraction.

List of Figures

List of Tables

Bibliography

[BCC$^+$03] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Bounded Model Checking*. 2003.

[BDW17] D. Beyer, M. Dangl, and P. Wendler. *A Unifying View on Smt-based Software Verification*. Autom Reasoning, 2017.

[BEK11] D. Beyer and M. Erkan Keremoglu. *CPAchecker: A Tool for Configurable Software Verification*. 2011.

[Bey20] D. Beyer. *Advances in Automatic Software Verification: SV-COMP 2020*. In Armin Biere and David Parker, editors, Tools and Algorithms for the Construction and Analysis of Systems, pages 347–367, Cham, 2020. Springer International Publishing.

[BF16] D. Beyer and K. Friedberger. *A Light-weight Approach For Verifying Multi-threadedprograms With Cpachecker*. 2016.

[BHT07] D. Beyer, T. A. Henzinger, and G. Théoduloz. *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis*. In Werner Damm and Holger Hermanns, editors, Computer Aided Verification, pages 504–518, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[BHT08] D. Beyer, T. A. Henzinger, and G. Theoduloz. *Program Analysis with Dynamic Precision Adjustment*. In 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pages 29–38, Sep. 2008.

[BKW10] D. Beyer, M. E. Keremoglu, and P. Wendler. *Predicate Abstraction with Adjustable-Block Encoding*. In Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, FMCAD '10, page 189–198, Austin, Texas, 2010. FMCAD Inc.

[BL12] D. Beyer and S. Löwe. *Explicit-Value Analysis Based on CEGAR and Interpolation*. CoRR, abs/1212.6542, 2012.

[BLW15] D. Beyer, S. Löwe, and P. Wendler. *Benchmarking and Resource Measurement*. 2015.

[CGSS13] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. *The MathSAT5 SMT Solver*. In Nir Piterman and Scott Smolka, editors, Proceedings of TACAS, volume 7795 of *LNCS*. Springer, 2013.

[CJA12] J. Christ, Hoenicke J., and Nutz A. *SMTInterpol: An Interpolating SMT Solver*. In SPIN, volume 7385 of *Lecture Notes in Computer Science*, pages 248–254. Springer, 2012.

[ES03] N. Een and N. Sörensson. *An Extensible SAT-solver*. 2003.

[IT20] O. Inverso and C. Trubiani. *Parallel and Distributed Bounded Model Checking of Multi-threaded Programs*. 2020.

[ITF$^+$14] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. *Lazy-CSeq: A Lazy Sequentialization Tool for C*. In Erika Ábrahám and Klaus Havelund, editors, Tools and Algorithms for the Construction and Analysis of Systems, pages 398–401, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[Löw17] S. Löwe. *Effective Approaches to Abstraction Refinement for Automatic Software Verification*. 2017.

[Wen17] P. Wendler. *Towards Practical Predicate Analysis*. 2017.