# TOWARDS SCALABLE SOFTWARE ANALYSIS USING COMBINATIONS AND CONDITIONS WITH CPACHECKER

by

Mehmet Erkan Keremoglu

B.Sc., Koç University, Istanbul, Turkey, 2005

M.Sc., Koç University, Istanbul, Turkey, 2007

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy
in the
School of Computing Science
Faculty of Applied Sciences

© Mehmet Erkan Keremoglu, 2011
SIMON FRASER UNIVERSITY
Fall 2011

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced, without authorization, under the conditions for "Fair Dealing." Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

## APPROVAL

**Name:** Mehmet Erkan Keremoglu

**Degree:** Doctor of Philosophy

**Title of Thesis:** Towards Scalable Software Analysis using Combinations and Conditions with CPAchecker

**Examining Committee:** Dr. Joseph G. Peters
Chair

Dr. Dirk Beyer, Senior Supervisor

Dr. Uwe Glässer, Supervisor

Dr. Robert D. Cameron, SFU Examiner

Dr. Helmut Veith, Professor, Faculty of Informatics
Technische Universität Wien
External Examiner

**Date Approved:** _____

## Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the "Institutional Repository" link of the SFU Library website (www.lib.sfu.ca) at http://summit/sfu.ca and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, British Columbia, Canada

revised Fall 2011

# Abstract

Verification of large-scale programs is a challenging problem. Software analysis tools focus on making a verification task both precise and efficient. We implemented the software analysis tool CPAchecker based on the configurable program analysis framework. CPAchecker aims to make it possible for a user to integrate and configure an analysis technique to the tool easily.

In order to handle large-scale programs, software analysis techniques use abstraction which introduce unsoundness and incompleteness to the process. If the sources of unsoundness and incompleteness is defined and evaluated by the user, the analysis will still be useful for finding potential bugs in the program. In our work, we used two related mechanisms to define and control potential sources of imprecision: Verification assumptions and verification conditions. Assumptions can be represented as a state predicate (a logical formula) that specify under what assumption an abstract state was reached during the analysis. Verification conditions are used as commands to guide the tool when some specific behaviour is observed during the analysis. In this thesis, we use verification assumptions and conditions to guide the analysis process. We added functionalities to CPAchecker such that it provides three mechanisms: (i) Starting the analysis task with a set of specified assumptions to run a partial verification. (ii) Using conditional analysis to increase the coverage of analysis and generating an analysis report encoded as assumptions even in cases where the conventional model checking fails to handle. (iii) Restarting the analysis with a set of different conditions and assumptions or deploying another analysis technique using an analysis report that has been already generated.

We extended CPAchecker to include a sequential composition framework that restarts the analysis with a new configuration using the result of another analysis. In our implementation of CPAchecker, the process of assumption and verification condition generation is

automated. Experimental results have shown us that using the configurable program analysis concept and guiding the analysis using assumptions and verification conditions was able to find bugs that traditional techniques were not able to identify, and increase the efficiency and precision of the analysis.

*To my newborn niece, Duru*

# Acknowledgments

I am heartily thankful to my supervisor Dr. Dirk Beyer for his endless support, understanding and encouragement. His research abilities, ideas, and deep knowledge of the topic was always there to help me to complete this thesis. He has made available his support in a number of ways, not only in teaching and in research, but by helping to solve any problem that I face during my 4 years at SFU. I also would like to thank Dr. Uwe Glässer from my supervisory committee who helped a lot with his feedback during my studies. Ideas and feedback of Dr. Tom Henzinger was immensely important for us, and having him as a research collaborator made it possible to have this thesis written.

I am grateful to my colleague and one of the main contributors of CPAchecker, Philipp Wendler who started working in the CPAchecker team since 2009. He has always been a great source of help. Many people have contributed to our tool CPAchecker, in particular, Alberto Griggio, Andreas Holzer, Stefan Löwe, Gregor Endler, Alexander von Rhein, Michael Tautschnig, and Grégory Théoduloz. We would not be able to have a robust and reliable tool without their contributions.

I would like to thank to the members of my thesis committee, Dr. Joseph G. Peters, Dr. Dirk Beyer, Dr. Uwe Glässer, Dr. Robert D. Cameron, and Dr. Helmut Veith for their time and feedback. I thank to Andreas Stahlbauer for proof reading the thesis.

My friends, Ashgan, Chirag, Philippe, Pınar, Recep made my life in Vancouver much easier and it was a pleasure to know them. Staff at SFU, Jeanie, Janet, Jason, Val, and Gerdi were always helpful whenever we needed something. I am grateful to Dirk, Philipp, Stefan, Max, Eva, and Malte for helping me to survive in my long visits to Germany, it was a great experience thanks to them.

This research would not have been possible without generous funds of Canadian NSERC.

Finally, I would like to express my love and gratitude to my beloved parents and my brother for their endless support and patience even when I was living very far away from them.

# Contents

**Approval** **ii**

**Abstract** **iii**

**Dedication** **v**

**Acknowledgments** **vi**

**Contents** **viii**

**List of Tables** **xi**

**List of Figures** **xii**

**List of Algorithms** **xiv**

**Nomenclature** **xv**

**1 Introduction** **1**

1.1 Contributions . . . . . 6

1.2 Structure of the Thesis . . . . . 7

**2 Background** **9**

2.1 Preliminaries . . . . . 9

2.1.1 Programs . . . . . 9

2.1.2 An Example Program and its CFA Representation . . . . . 10

2.1.3 Reachability Problem . . . . . 12

2.2 Automated Techniques for Software Analysis . . . . . 14

2.2.1 Static Analysis by Abstract Interpretation . . . . . 15
2.2.2 Abstract Interpretation Example . . . . . 17
2.2.3 Software Model Checking . . . . . 21
2.2.4 Predicate Analysis . . . . . 22
2.2.5 Predicate Analysis Example . . . . . 24
2.2.6 Abstract Reachability Tree . . . . . 24
2.2.7 Counterexample-guided Abstraction Refinement . . . . . 26
2.3 Configurable Program Analysis . . . . . 28
2.3.1 Definition of CPA . . . . . 29
2.3.2 Configurable Program Analysis with Dynamic Precision Adjustment . 30
2.3.3 CPA Formalism . . . . . 31
2.3.4 CPA Algorithm . . . . . 34
2.4 Related Work: Automated Software Analysis Tools . . . . . 35

**3 Implementation of CPA Framework:** CPAchecker **40**
3.1 CPAs for the Analyses Used in the Thesis . . . . . 41
3.1.1 CPA for Location Analysis . . . . . 41
3.1.2 CPA for Explicit Analysis . . . . . 41
3.1.3 CPA for Adjustable-block Encoding . . . . . 43
3.2 Architecture and Implementation . . . . . 48
3.3 CPAchecker Extensions . . . . . 51
3.3.1 Algorithm Interface and CBMC Feasibility Check . . . . . 52
3.4 Experiments with CPAchecker . . . . . 55
3.4.1 Benchmark Programs and Configurations . . . . . 55
3.4.2 Experiments for Comparison with Other Tools . . . . . 57

**4 Sequential Composition of CPAs** **64**
4.1 Behaviour of Different Analysis Techniques . . . . . 64
4.1.1 Imprecision of Explicit Analysis . . . . . 65
4.2 Implementation of a Sequential Composition Framework . . . . . 67
4.3 Experiments Using Sequential Composition . . . . . 70
4.4 Discussion . . . . . 77

**5 Verification Assumptions and Conditions** **79**
5.1 Motivation . . . 80
5.1.1 Examples . . . 82
5.1.2 Contributions and Applications . . . 84
5.2 Verification Assumptions . . . 86
5.2.1 Conditional Model Checking and Assumptions as Output . . . 89
5.2.2 Formalization and Implementation of Assumptions . . . 92
5.3 Verification Conditions . . . 96
5.3.1 Failure Classes and Failure-preventing Conditions . . . 97
5.3.2 Formalization and Implementation . . . 101
5.4 Combining Verification Conditions and Assumptions . . . 103
5.4.1 Composite CPA for Conditions and Assumptions . . . 104
5.5 Verification Conditions for Bug Detection . . . 110
5.6 Adjustable Conditions . . . 111
5.6.1 Motivation . . . 111
5.6.2 Algorithm and Implementation . . . 112
5.6.3 Threshold Adjustment for Conditions . . . 116
5.7 Experiments for Adjustable Conditions . . . 117
5.7.1 Adjustable Conditions on Benchmark Programs . . . 117
5.7.2 Adjustable Conditions for Improving Coverage . . . 123
5.8 Using Output of one Algorithm as Input for another Algorithm . . . 125
5.8.1 Example . . . 125
5.9 Experiments for Using Output of an Algorithm as Input . . . 130
5.10 An Efficient Combination and Discussion . . . 131
5.11 Related Work . . . 138

**6 Discussion** **140**
6.1 State-of-the-Art in Software Verification . . . 140
6.2 Conclusion . . . 141
6.2.1 Contributions . . . 142
6.3 Future Work . . . 143

**Bibliography** **145**

# List of Tables

3.1 Comparions with other tools . . . . . 58

4.1 Using sequential composition . . . . . 71

5.1 Representations of conditions in an assumption formula . . . . . 108
5.2 Examples for using conditions with explicit analysis on `diskperf_simpl1` . . 109
5.3 Conventional versus conditional analysis on programs with a bug . . . . . 111
5.4 Adjustable conditions . . . . . 118
5.5 Comparison of the size of reached set using adjustable conditions . . . . . 123
5.6 Experiments with assumption automata as condition . . . . . 132
5.7 Combinations of 5 different configurations . . . . . 134

# List of Figures


2.1 Example C program . . . . . 11
2.2 CFA for the program shown in Fig. 2.1 . . . . . 13
2.3 Example program with a loop . . . . . 17
2.4 CFA for the example in Figure 2.3 . . . . . 18
2.5 Simulating the run for example in Figure 2.3 . . . . . 19
2.6 Lattice for explicit value tracking . . . . . 20
2.7 Lattice for parity domain . . . . . 20
2.8 Abstract interpretation based on parity for the example in Figure 2.3 . . . . . 21
2.9 List of predicates for the predicate analysis of the example program shown in Figure 2.3 . . . . . 24
2.10 Predicate analysis process for the working example . . . . . 25
2.11 Example ART for the analysis explained in Figure 2.10 . . . . . 27

3.1 CPAchecker — Architecture overview taken from [27] . . . . . 49
3.2 Interactions between CPAchecker components taken from [27] . . . . . 50
3.3 The algorithm interface is used by several algorithms . . . . . 53

4.1 Example program producing an imprecise result with the explicit analysis . . 66
4.2 CFA for the example in Figure 4.1 . . . . . 67
4.3 ARG for the example in Figure 4.1 . . . . . 67
4.4 Design Overview of Sequential Composition . . . . . 68

5.1 Example program with loop . . . . . 82
5.2 Example program with non-linear safety condition (taken from [19, 20]) . . . 83

5.3 Example state space for a program which is analyzed using a limiting condition on the length of the path which is taken from [19]. Limiting the length of the path lets the analysis to skip shaded part and find the error state shown with the brown-colored node. . . . . 85
5.4 Example C program . . . . . 87
5.5 ARG nodes for locations 5, 6 and 7 for the example given in Figure 5.4 when the analysis starts with using the assumptions $(l = 6) \Rightarrow ((value1 > 1000) \wedge (value1 < 10000))$ and $(l = 7) \Rightarrow ((value2 > 0) \wedge (value2 < 100))$ . . 88
5.6 A part of the CFA for the example program with loop given in Figure 5.1 . . 90
5.7 A part of the ARG for the example program with loop given in Figure 5.1 . . 91
5.8 Architecture overview of assumption collecting algorithm . . . . . 96
5.9 Example Conditions for Conditional Model Checking; column *Impl.* lists the conditions we implemented . . . . . 99
5.10 Architecture overview of assumption collecting algorithm using repeating locations monitoring CPA . . . . . 106
5.11 An example assumption in human-readable format for the condition $c_{24}$ in Table 5.2 . . . . . 110
5.12 CPA with progress observer CPA including *Repeating Locations*, *Path Length* and *Time-out*, and assumptions storage CPA. . . . . 114
5.13 CFA for the example program given in Figure 5.2 . . . . . 127
5.14 ARG for the example program with the CFA in Figure 5.13 after predicate analysis with the predicate $(i \geq 1000000)$ followed by a precise path analysis . 128
5.15 Assumption automaton for the example program . . . . . 129
5.16 Flow of the configurations . . . . . 133

# List of Algorithms


1 $CPA(\mathbb{C}, R_0, W_0)$ . . . . . 35
2 $AnalysisWithFeasibilityCheck(A_1, A_2, \mathbb{C}, R_0, W_0)$ . . . . . 54
3 $SequentialComposition(L)$ . . . . . 69
4 $AssumptionCollecting(A, \mathbb{C}, R_0, W_0)$ . . . . . 94
5 $AdjustableConditions(A, \mathbb{C}, R_0, W_0)$ . . . . . 113

# Nomenclature

| | |
|---|---|
| ART | Abstract Reachability Tree |
| CFA | Control Flow Automata |
| CEGAR | Counterexample-Guided Abstraction Refinement |
| LBE | Large-Block Encoding |
| ABE | Adjustable-Block Encoding |
| SBE | Single-Block Encoding |
| BDD | Binary Decision Diagram |
| CPA | Configurable Program Analysis |
| SMT | Satisfiability Modulo Theories |
| CEX | Counter example |
| SAT | Satisfiability |
| BMC | Bounded Model Checking |
| CMC | Conditional Model Checking |

# Chapter 1

# Introduction

Complex software systems have become very commonly-used in recent years. With the technological developments in hardware, more complex systems can be developed and they are used almost for everything nowadays. Software systems are crucial parts of the critical systems and correctness is one of the major factors of their quality. Evolving as more complex components and interacting with different environments, checking the correctness of software is a challenging question. Actually, it has become one of the main challenges of computer science research [75]. Testing and debugging are still predominant methods for detecting and fixing problems that lead to unexpected behaviour, which are called as bugs, in the software but the ever-growing complexity of software systems has become a bottleneck for their success. With infinitely many possible input and nondeterministic interactions with different components, testing is prone to miss potential bugs in the program. Researchers used formal verification methods to reason about the behaviour of programs with any input of the program. Formal verification research dates back to works of Hoare and Floyd [62,74]. These initially proposed techniques which are closely related with theorem proving are manual efforts that is ideal for small-scale software but they also led to the development of practical methods that try to analyze the systems exhaustively. Developments in hardware verification also inspired many techniques and optimizations to be developed for software analysis.

A technique that is widely used for verification of software is reachability analysis, which is the main focus of this thesis. It is the problem of checking the possibility of reaching to an error state, a bug, in a software system. The focus of software analysis research is to provide techniques that are scalable to real, industrial-size code, to be practical and to

require minimum user-interaction [87]. We are interested in techniques that are automated and practical. Automated analysis techniques can be classified in two categories which is based on their verification targets.

The first set of analysis techniques, called static analysis, over-approximates the possible states of a software system and extracts information about the possible behaviour that can be observed during the execution. The reachability problem is an undecidable problem since it can be reduced to the halting problem. Therefore, static analysis techniques use approximations and heuristics to be practically useful. The possible states of a software system can be represented over a set of pre-defined elements. The choice of these elements determines how the analysis will process and how the system will be analyzed. For example, all integer variables in a program can be represented as positive or negative integers. This procedure is called abstraction and since it represents more elements than the actual state of the program, it is an over-approximation of possible behaviour [51]. Using the flow of the program, these abstract elements are updated and the error states are checked for reachability. Examples of error states are assertion failures, possible null dereferences, accessing arrays out of bounds, division by zero operations or any other user-defined conditions. Similar techniques to static analysis are also used for compiler optimizations and historically, they are known for their efficiency. The efficiency comes with the problem of unsoundness: Due to the over-approximation, they report some bugs that are not feasible to be reached during the execution of the program.

The second set of analysis techniques, called software model checking, is the process of creating a model of the program and validating the satisfiability of a specification by that model [43]. The specification is usually a formula that can represent the absence of a bug condition or a property that the system should hold such as executing some operations in some pre-defined order. The model checking techniques and static analysis techniques can be formalized with the same components but there is a historical difference between them. Model checkers try to build the state space of the program exhaustively rather than over-approximating for efficiency. This makes model checking more precise, reporting fewer bugs that are not actually reachable, and providing a counter-example which is a replay of the error state which makes the process highly reliable. The drawback of software model checking is the efficiency since it may not be practical to build the entire state space of large-scale programs.

Configurable program analysis (CPA) is a recent concept that uses a common formalism to express both static analysis and model checking [24]. Using the common components, it aims to close the gap between static analysis and model checking. A software analysis framework can combine several techniques that are based on static analysis and model checking and it can adjust the common operators that run the analysis to be used together. Such an adjustment lets the user to achieve better precision and efficiency for the analysis. For example, an analysis may use a model checking approach to have information about some parts of the program by exhaustively exploring all possible states but can switch to a static analysis approach for some other parts and use over-approximation to be efficient. Intermediate settings for common components of several analyses is shown to be useful for achieving practicality [24].

In this thesis, we present a tool for configurable software analysis. CPAchecker is a software analysis tool for C programs which is developed in Java [27]. CPAchecker provides the common components specified by configurable program analysis as interfaces to express different methods and makes it possible to integrate an analysis to the tool easily. It also provides components that can combine different analyses and makes it possible to configure the settings of the analysis easily. Providing the basic reachability algorithm and common components for implementing the reachability analysis, we extended CPAchecker to be practically useful for analysis of real programs. We already implemented several program analysis techniques with different properties including model checking techniques that are flexible and configurable to achieve efficiency without losing precision [17, 27, 28]. Using an algorithm interface which is compatible with the main reachability algorithm of the tool, we can extend the analysis to support many practical methods to be used flexibly without changing the implementations for configurable program analysis. For example, since our tool does not support operations at bit-level currently, we provide an algorithm that makes a call to external tools that runs a precise bit-level analysis to check whether a reported bug from CPAchecker analysis is indeed feasible. This algorithm can use the main reachability algorithm as a component. Being loyal to configurable program analysis' compatibility and flexibility, CPAchecker can be used for many applications of software analysis.

A practically useful extension to CPAchecker is a framework that supports combining different analysis sequentially [19, 20]. The initial idea behind configurable software analysis was using several software analyses together by configuring intermediate settings for them, thus running them in parallel. We extended the definition of combinations to a sequential

combination algorithm: The analysis will take several analyses as input and run them one after another without terminating after an analysis cannot finish the search. This approach is even useful in its basic form: Some analyses that inherit from the static analysis approach are designed to be efficient and they can be limited by some resources such as time or memory hoping that they will find a bug very fast; and if they are not successful, we switch to a more precise analysis and run an exhaustive search. We also used the output of one analysis for improving the results of another analysis in the sequential composition framework.

As explained before, the reachability problem is undecidable. Thus, some techniques produce imprecise results in order to terminate after a reasonable time is spent: They conclude the search before exploring the entire state space of the program. This makes the process less reliable but many techniques have to take this approach to be practical. If the reason of the imprecision is captured by the tool in some form, we can later use it to produce sound results. In order to utilize imprecise results, we introduced two related concepts for software analysis: Verification assumptions and verification conditions [19, 20].

Verification assumptions are logical formulas that define under which assumptions the analysis was conducted. For example, if the analysis outputs an imprecise result, an assumption can define why the analysis was not precise by limiting the explored states of the program by an assumption. This is useful in several ways: First, many analysis techniques make implicit assumptions, such as ignoring overflows of variables; defining relevant assumptions, the user can define these cases explicitly. Second, if an analysis cannot finish the search in reasonable limits using given resources, it can terminate safely and summarize the work done so far by generating an assumption. We provide an assumption collecting algorithm in CPAchecker for generating assumptions during the analysis. A generated assumption can be useful in many ways: Summarizing the work done, it can give clues about the structure of the analyzed program or the reason of the imprecise result of the analysis.

Verification conditions are configurations to guide the search algorithm. A verification condition can determine whether to continue exploring some parts of the state space or stopping the analysis for that part and continue with different verification targets. We have provided several conditions and implemented several of them as CPAs to be used with our static analysis engine. By observing the behaviour of the program, the tool can use predefined conditions to run the state space search. Combining with assumptions, verification conditions can be useful in getting results for programs that normally do not terminate within given resources. For example, a condition that observes the time spent or memory

consumed can terminate the analysis when the resource usage reaches to critical limits and summarize the work done so far with a generated assumption. The analysis, otherwise, would crash with no useful results.

We also used verification assumptions and verification conditions together with our sequential composition framework. In the first setting; we used sequential composition to adjust conditions given to the algorithm to have a more efficient search. Such a configuration starts the analysis with no conditions. Then, after running the analysis for a limited time (possibly a very short time) and observing the progress, it generates a condition which might be useful to reach to an error state or to cover a bigger part of the program's state space. The analysis is, then, restarted with this condition and run for a limited time again. If the analysis does not terminate within the time limit, it adjusts the condition again and continues the search. This framework, which is called adjustable conditions framework was useful especially in detecting bugs efficiently since we could automatically adjust the search strategy based on observations from the previous runs of the analysis. It also introduces improvement for discovering more facts about the behaviour of the program since it tends to skip parts that usually consume significant resources.

Another mechanism for increasing efficiency of software analysis by CPAchecker was to use assumptions which were generated by one technique as an input for another technique. This was easily achievable inside the sequential composition framework. The idea behind this approach uses characteristic strengths of different techniques to handle specific properties of programs. If a part of the program is more suitable for analyzing it with a particular technique and some other part is analyzed more efficiently with a different technique, this approach is ideal to run an efficient analysis. We implemented the approach in the following way: First we run the analysis using a particular technique that explores a part of the possible states of the program. If it cannot handle the task of exploration of other parts, we terminate the analysis and generate an assumption that summarizes the work done by that particular technique. Then, the sequential composition algorithm switches to another technique but since we have already explored some part of the state space, the tool can skip the verified part and concentrate on searching the unexplored parts. This is a complementary method for exploiting strengths of several techniques and it was able to analyze some programs by CPAchecker which were not being analyzed successfully by other tools and techniques. Assumptions can be used as a common format for different analyses and

become equally useful as input and output for the algorithm. They provide communication between different techniques.

In this thesis, we focus on providing methods for practical software analysis. CPAchecker was a first step for achieving our goals: It is configurable, extendable and flexible. An analysis technique can be described as a CPA and can be implemented in CPAchecker simply by extending given interfaces. It is also possible to use the basic reachability algorithm of CPAchecker as a component for many practical purposes also by simply using the algorithm interface. In addition to its core parallel composition framework, we also added a sequential composition framework to CPAchecker. Sequentially composing different methods helped us choosing different limits and configurations for different analyses in order to achieve better efficiency without loosing precision. Introducing verification conditions and assumptions helped us to use imprecise results from an analysis effectively. We have shown that we have improved the success of CPAchecker significantly using sequential composition, verification conditions and verification assumptions. Configuring and selecting suitable conditions for a program analysis task, CPAchecker was able to analyze many programs successfully from several benchmarks that are used in software analysis research.

## 1.1 Contributions

1. We have designed and developed the software analysis tool CPAchecker. CPAchecker provides interfaces that is a one-to-one representation of configurable analysis concept. We experimented on benchmark programs that were used for evaluating software analysis tools and provided implementations for commonly used techniques. Among them, there is an implementation of a highly efficient and configurable model checking technique as well.

2. We extended CPAchecker to run the analysis efficiently with a minimum number of false results. For that, different tools that can run more precise analysis are included as compatible algorithms with the main reachability algorithm of the tool. Integration of configurable options and algorithms improved the analysis results significantly.

3. A sequential composition framework is implemented and used. We have shown that using different combinations of several analyses can improve the analysis results. Classifying different techniques based on their efficiency, precision, and performances, we

can either use combinations of different analyses in parallel or sequentially to practically check correctness of software. This also includes the ability to insert analysis by other tools into CPAchecker as a component and use their results.

4. We defined and tested verification conditions which can improve bug hunting abilities of the analysis, as well as improving the coverage of the verification. We implemented several conditions that can be configured for improving efficiency. Combined with the sequential composition framework, CPAchecker can adjust these conditions automatically to have better results.

5. Verification assumptions are used to rule out 'no results' cases from the model checking process. They are used to define the sources of the unsoundness and give meaningful results to the user in cases where traditional approaches were returning no results. Combining with the verification conditions, assumptions are used to guide and improve the analysis. We also used them as common output and input for several types of analyses, thus allowing to use results from one analysis for improving the results of another.

## 1.2 Structure of the Thesis

Chapter 2 presents formalism for software systems that will be used throughout the thesis and introduces static analysis and software model checking that are implemented in CPAchecker. We also list some software analyses tools and explain which methods or techniques they used for being practical. In chapter 2 we define configurable program analysis (CPA) and gives formal definitions for building a CPA.

Chapter 3 first defines analyses that are used in this thesis in CPA formalism and introduce large-block encoding (LBE) approach for predicate abstraction which is an efficient model checking approach. Then, it explains the design and implementation of CPAchecker and explains how it is extended for handling practical cases. We briefly describe the structure of the tool and show its interactions with other tools to improve verification results. It also gives experimental results for comparing its basic analysis algorithms with the state-of-the-art analysis tools.

Chapter 4 introduces the sequential composition framework and shows how it can be utilized to achieve better results with CPAchecker. We also discuss the design issues to

extend CPAchecker to support sequential compositions of several analyses in Chapter 4. Finally, we show how CPAchecker results are improved using sequential composition by presenting experimental results.

Chapter 5 introduces verification assumptions and verification conditions that are used to handle imprecise results of software analysis. Starting with explaining the motivation, Chapter 5 progresses by giving definitions for verification conditions and verification assumptions and explain how they were implemented in CPAchecker. It shows how they are combined together to run the analysis efficiently. We also explain how the sequential composition framework is used with assumptions and conditions to provide communication between different analyses. Lastly, we present experimental results that show the improvement using these concepts.

Chapter 6 concludes the thesis and gives future research directions.

# Chapter 2

# Background

In this chapter, we introduce notations that are used to explain algorithms and techniques throughout this thesis. The software analysis problem that we try to solve is also defined in this chapter. Finally, we introduce analysis techniques that we used in our tool and provide a short survey about other tools using these techniques.

## 2.1 Preliminaries

In this section, we provide a common representation for the programs that we analyze and define the software analysis problem.

### 2.1.1 Programs

We use a simple imperative programming language for representation that has assignments and assume operations over integer variables. Our representation language can also model loops, goto statements specified by labels and function calls with no side-effects.

A *program* is represented by a *control-flow automata* (CFA). The nodes of the CFA model program locations that correspond to program counter $pc$, which is denoted by the set $L$. The initial program location is $pc_0$ which corresponds to the program entry point. The set $G \subseteq L \times Ops \times L$ of control-flow edges model the operations of the program. $Ops$ is the set of all possible operations supported by the language. An operation (an edge in the CFA) moves the program from one program location to another program location. A program is represented by a CFA $A = (L, G)$ and a program entry location $pc_0$.

The set $X$ denotes the program variables that occur in operations from $Ops$. A *concrete state* $c : X \mapsto \mathbb{Z}$ assigns variables from $X$ to integer values. The set of all concrete states of a program is denoted by $C$. A *region* $r \subseteq C$ is a set of concrete states. The edges of the CFA define a transition relation $\xrightarrow{g} \subseteq C \times \{g\} \times C$. The entire set of control-flow edges is defined as the union over all edges $\rightarrow = \bigcup_{g \in G} \xrightarrow{g}$. If $(c, g, c') \in \rightarrow$, then there is a transition from $c$ to $c'$ with the control-flow edge $g$ and we write $c \xrightarrow{g} c'$ to represent this relation. If there is a $g$ with $c \xrightarrow{g} c'$, then we write $c \rightarrow c'$. In the concrete program, $c \xrightarrow{g} c'$ corresponds to the execution of the operation represented by $g$ which moves the program to state $c'$ from the program state $c$.

Reachability is defined over concrete states of the program. A concrete state $c_n$ is reachable from a region $r$, denoted by $c_n \in Reach(r)$, if there exists a sequence of concrete states $\langle c_0, c_1, \ldots, c_n \rangle$ such that $c_0 \in r$ and for all $1 \leq i \leq n$, we have $c_{i-1} \rightarrow c_i$. A reachable path is also called as a feasible path. A reachable path corresponds to a path on the program which can be taken by executing the corresponding operations.

### 2.1.2 An Example Program and its CFA Representation

In Figure 2.1, we present an example C program to introduce the concepts we use in this thesis. The example code has three `int` variables: `value1`, `value2` and `max`. At lines 7 and 8, we call two different functions that return a random integer value and assign the return value of `randFunc1()` to `value1` and the return value of `randFunc2()` to `value2`. Note that the user might provide an implementation for both functions or they might be considered as external calls. Each CPA provides its own implementation for handling an external call. There are multiple mechanisms to handle external calls if we know the range of their return values which will be covered in following chapters. For the sake of simplicity, we assume that calls to random functions are considered as external calls in this example.

Lines 10 to 16 compare the values of `value1` and `value2` and assign the maximum of the two elements to the variable `max`. Lines 18 to 22 ensure that `max` has been assigned to the value of either `value1` or `value2` and if it is not, it jumps to the ERROR label and returns -1. If `max` holds the value of one of the two values at the end of the procedure, the procedure returns 0.

Note that this small piece of code ignored the case where `value1` and `value2` have the same value. In that case, there is no assignment to `max` and it might have any value as an uninitialized variable which might lead to unforeseen problems when using the program

```
int main(void)
{
  int value1;
  int value2;
  int max;

  value1 = randFunc1();
  value2 = randFunc2();

  if(value1 > value2){
     max = value1;
  }

  else if(value2 > value1){
     max = value2;
  }

  if(max != value1){
    if(max != value2){
      goto ERROR;
    }
  }

  END:
  return (0);
  ERROR:
  return (-1);
}
```

Figure 2.1: Example C program

code. If the ranges of the return values for `randFunc1()` and `randFunc2()` are large enough, this error might be difficult to detect using traditional testing techniques.

In Figure 2.2, we show the CFA for the example in Figure 2.1. Node numbers represent the program counter *pc* and the edges represent the operations. The edge from 'Node 1' to 'Node 2' is to represent the initialization of the program and following nodes represent declarations, assignments, function calls, return statements, goto statements and assume operations for the program in Figure 2.1. For example, the edge from 'Node 5' to 'Node 6' corresponds to the assignment at line 7 of the code. We show branching points with diagonal shapes in the CFA representation. The control flow has conditional statements (which we also refer as assume operations) and nodes 7, 10, 8 and 16 represent these operations. They all have two outgoing edges. For example 'Node 7' represents the assume operation at line 10. The program follows the edge to 'Node 9' if (`value1` > `value2`), and follows the edge to 'Node 10' if not.

We use CIL to pre-process the given program [97]. CIL is a tool that transforms a C code to C intermediate language and many complex operations are simplified by conserving the behaviour of the program. For example, if there is an `if{x}-else if{y}-else{}` branch in the code, it normally produces three outgoing edges for the assume operation edge. CIL transforms these statements into the form `if{x}-else{if{y}-else{}}`, so instead of one node to represent the assume operation, there are two of them and they both have two outgoing edges. This can also be seen on our example CFA in Figure 2.2 for $7 - 9$, $7 - 10$, $10 - 12$, and $10 - 13$ edges. This conversion guarantees that a node representing an assume operation has at most two outgoing edges.

### 2.1.3 Reachability Problem

In this thesis and with CPAchecker, we usually run a reachability analysis. Intuitively, reachability analysis checks if a program location in the code is reachable with the given initial location and state. Throughout the thesis, we will use the 'ERROR label' to define a state which indicates a potential bug in the program (or an assertion which also indicates an error condition). Any user given annotation can be converted to an 'ERROR label'. For example

```
assert(y);
```

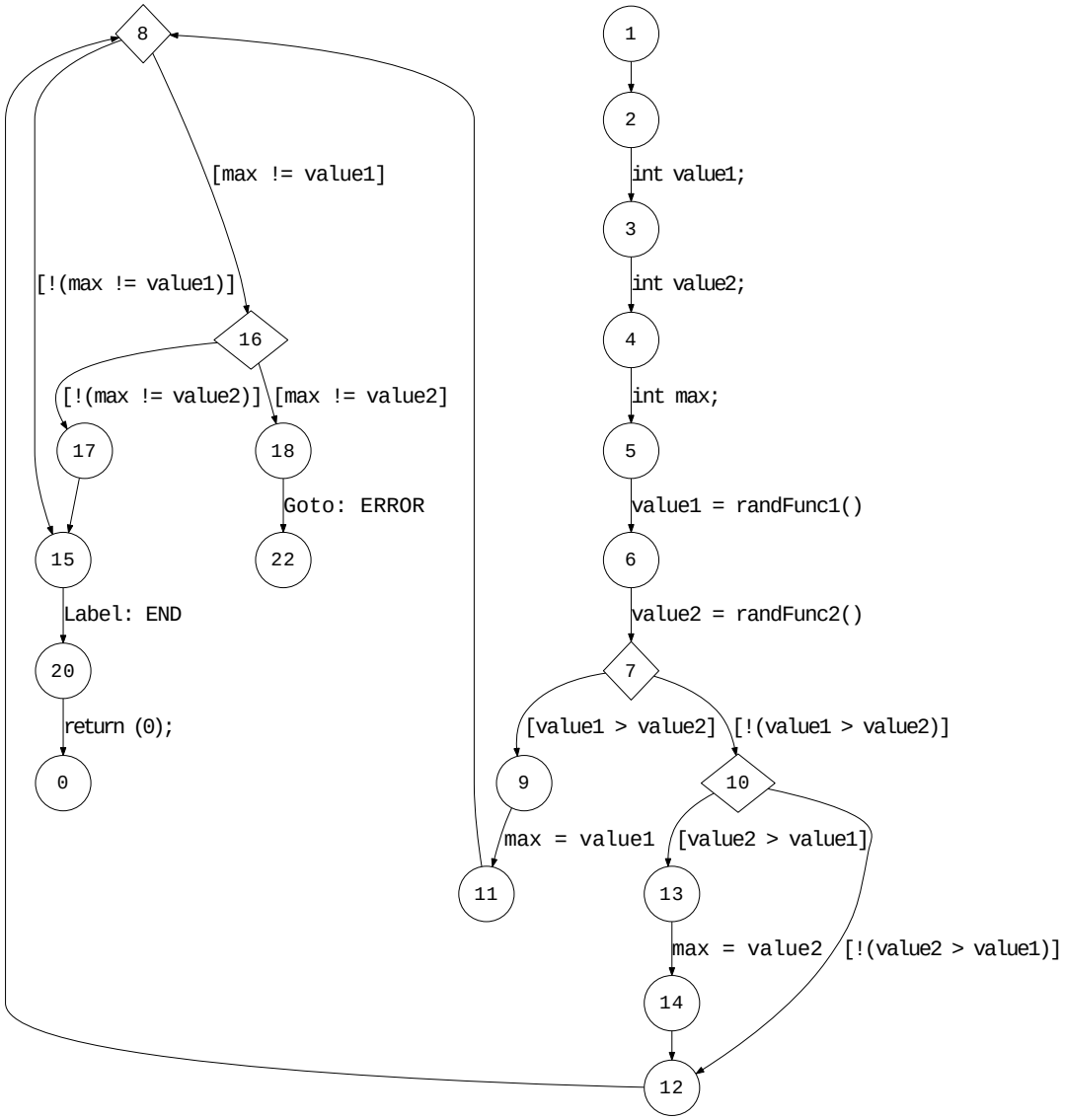


Figure 2.2: CFA for the program shown in Fig. 2.1

can be converted to

```
if(!y) {
   goto ERROR;
}
```

This conversion helps us to define many safety properties as a reachability problem and use existing tools on a rich set of programs. For example, to check if all division operations in a program are safe (i.e., the code does not perform division by zero), we can simply insert assertions which check if the divisor can be 0 before each division operation in the program.

As we defined reachability in Section 2.1.1, reachability to an 'ERROR label' is defined on the reachable paths of the program. The set $ERR$ denotes the set of 'ERROR' states in the program. Any state computed for an 'ERROR label' location is in the set $ERR$. An 'ERROR state' is reachable if $\exists\ c_{err} \in Reach(r_{init})$ where $r_{init}$ is the initial region and $c_{err} \in ERR$.

The reachability problem for software analysis can be reduced to the halting problem for Turing-complete languages [89]. This makes the problem undecidable and not all analyses are guaranteed to terminate. Techniques that we explain and use in this thesis aim to be sound but incomplete. Being sound and incomplete suggests that the tools will report SAFE if the specified error condition is not present in the program. On the other hand, it is possible that the tool reports UNSAFE which means that the error condition is detected in the program, and in some cases it may represent a non-reachable concrete state. Later, we propose some techniques which also sacrifice soundness to be practical but we want to handle these cases and we want to use the sources of the unsoundness to have more efficient and precise analysis.

## 2.2 Automated Techniques for Software Analysis

In this section, we introduce commonly used automated software analysis techniques. We specify these techniques as automated techniques because there are many verification methods that rely on continuous user-interaction (especially formal verification tools) and we are interested in tools that require minimum user-interaction and can be run automatically.

Software systems are nowadays widely used in everyday life and correctness of some systems is of vital importance. Most of the practically used systems are very large in size

and have interactions with many other software and hardware components. This makes it very difficult to detect and fix possible errors and an unforeseen behaviour may lead to catastrophic results [1].

Testing and debugging are still the most commonly used methods for detecting software errors. The main drawback of these dynamic methods is that they execute the program to test the behaviour of the system and it is usually impossible to cover all cases. On the other hand, software verification tools aim to capture how the system will react with any input of the software. This is very costly, so it is impossible to check for total correctness of the program with formal methods too. Several techniques have been developed that aim to verify partial correctness of the system such that instead of trying to prove total correctness, they can be configured to check some properties of the software.

A verification tool should be fast, usable with limited resources and precise in the results. We explain two automated software verification techniques that run the analysis on the source code and try to check the state space of a program. The first technique is static analysis that uses abstract interpretation. The basic characteristic of this technique is its efficiency and the ability to prove simple facts about the programs. The other technique, model checking, is more precise and can prove stronger safety properties about the program but it is more costly and usually slower. In the following chapters, we explain how these techniques are used by our tool CPAchecker.

### 2.2.1 Static Analysis by Abstract Interpretation

Static analysis by abstract interpretation (or simply static analysis) denotes a group of techniques that analyze the software without executing the code. Despite the broad scope of the definition, the research community in general refers to efficient techniques by static analysis. The basic aim of the analysis is to observe possible behaviour of the program and try to detect anomalies of the code.

Abstract interpretation was introduced by Cousot and Cousot [51]. A software system proceeds by assigning values to some variables (or memory locations) from a domain. An abstraction maps the values assigned to variables in concrete execution to values from another domain (possibly with a smaller space). The static analysis using abstract interpretation propagates using operations from the program flow and terminates when the set of abstract

[1] Check `http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=all`

elements saturate. The analysis saturates for an element when a fixpoint is reached for an operation on the program flow, i. e. all possible values that can be assigned to variables are captured by the abstraction and no new information can be added about the program at that location.

Abstraction can be done by several methods, but traditionally most of the abstraction techniques preserve soundness of the analysis. Due to undecidability of the problem, preserving the soundness and completeness of the analysis by developing some abstraction techniques while running the analysis efficiently became the central issue for static analysis tools. Abstraction techniques make it possible to observe the relevant information about the program. They produce an output state space that has parallel behaviour with the real program.

`Soundness` requires that abstract results given by the analysis are consistent with the concrete semantics of the program. In other words, the abstract program should never produce a `false negative`. A `false negative` is the case where the analysis claims that the program is safe with respect to the given property even though the concrete semantics contain an error. `Completeness` requires that the analysis does not produce `false positives` [43]. A `false positive` is the case where the analysis reports a spurious bug. Most of the static analysis tools preserve soundness and intend to minimize the number of false alarms [6].

**Abstraction domain.** An `abstract element` represents a set of concrete states of the program. These elements are from a set of elements that is the domain of the abstraction. Abstract states are ordered by a partial order, so they build a lattice and a concrete state of the program is mapped to an element of the lattice. Since abstract states represent a set of concrete states, they over-approximate the solution. Over-approximation preserves the soundness but it usually makes the analysis incomplete.

**Abstraction Function.** Abstraction provides a mapping from concrete states to abstract elements. The abstraction function provides this mapping.

**Merging.** Merging at join points, used together with abstract interpretation, lets the analysis terminate by reaching to the fixpoint. In typical static analysis that uses program CFA as its model, abstract states are merged at same program locations. Merging means to have a coarser abstraction. This property is compatible with the lattice structure of the abstract states. Whenever two states are merged, they produce an upper bound element on

```
1  int main() {
2    int temp;
3    int i = 0;
4    int j = 5;
5    int k = 0;
6
7    while(k < 10){
8      i = i - 4;
9      j = j + 2;
10     temp = j + i;
11     assert(temp != 0);
12     array[k] = temp;
13     k++;
14   }
15
16   return(0);
17 }
```

Figure 2.3: Example program with a loop

the lattice. If used with widening which means to produce a coarser upper bound element than the minimum upper bound, it guarantees to reach a fixpoint and that is the key for the termination of the analysis.

Since the structure of configurable program analysis is a one-to-one representation of abstract interpretation, we will formally define concepts that are used to describe abstract interpretation later.

### 2.2.2 Abstract Interpretation Example

A small example given in Figure 2.3 is used to illustrate how static analysis works. The corresponding CFA is shown in Figure 2.4. The CFA is simplified by only including CFA nodes that lead to an error label. Assignment operations which are consequently executed are summarized in one edge, such as the edges from 1 to 2 and from 3 to 4. The program simply tries to fill in an array with a value that is computed using variables `i` and `j` while updating the values of these variables inside a loop. The assertion at `line 12` validates that the program does not insert a zero to the array. In other words, the safety property in this program is not to have `temp` assigned to 0 at `line 12`.

**Simulating the Run of The Program Explicitly.** Since we know that this program proceeds by assigning integer values to integer variables, we can use an approach that

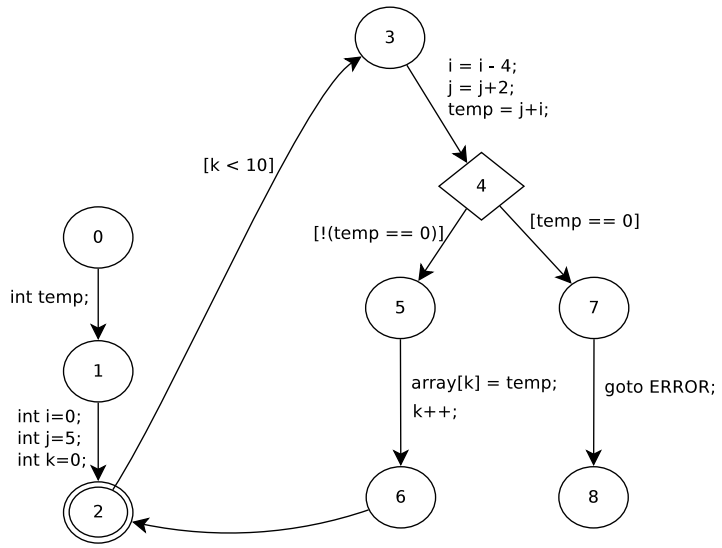


Figure 2.4: CFA for the example in Figure 2.3

simulates the execution of the program by explicitly tracking the values assigned to each variable with the flow of the program. An example of this operation is shown in Figure 2.5. In this figure, we show the values that are assigned to variables after each step of the algorithm. We first show the location of the CFA and the values for each variables in curly brackets. For example "Iteration 1) $1 \rightarrow 2: \ [2a, \ \{i = 0; \ j = 5; \ k = 0\}]$" states that after the first iteration of the simulation, the edge from location 1 to location 2 of the CFA is taken and for location 2, it is known that i is assigned to 0, j is assigned to 5 and k is assigned to 0. We append a letter to locations such as $2a$ or $2b$ to show that the same location is visited as the program proceeded and new values are assigned to variables. At iteration 5, the assume edge to location 7 is tested but it is not reachable because the condition in the assume operation is not satisfied. After the last iteration, the while loop terminates and the analysis stops. The execution sequence in Figure 2.5 is a replay of the concrete execution of the program. The entire state space is built in 56 iterations.

**Analyzing the Program with Abstract Interpretation.** Now, we can use an abstraction based analysis for checking the correctness of the program instead of tracking the execution explicitly. An intuitive example for abstract interpretation is an abstraction based on 'parity' of the assigned value of the variable. If an odd number is assigned to the variable, it is set to `ODD`; if an even number is assigned, it is set to `EVEN` and if the parity is not known it is set to `UNKNOWN`. With each statement on the CFA, the values of abstract

Iteration 0) : Initial State $[1, \{\}]$
Iteration 1) $1 \rightarrow 2$ : $[2a,\ \{i = 0;\ j = 5;\ k = 0\}]$
Iteration 2) $2 \rightarrow 3$ : $[3a,\ \{i = 0;\ j = 5;\ k = 0\}]$
Iteration 3) $3 \rightarrow 4$ : $[4a,\ \{i = -4;\ j = 7;\ k = 0;\ temp = 3\}]$
Iteration 4) $4 \rightarrow 5$ : $[5a,\ \{i = -4;\ j = 7;\ k = 0;\ temp = 3\}]$
Iteration 5) $4 \rightarrow 7$ : NOT REACHABLE
Iteration 6) $5 \rightarrow 6$ : $[6a,\ \{i = -4;\ j = 7;\ k = 1;\ temp = 3\}]$
Iteration 7) $6 \rightarrow 2$ : $[2b,\ \imath = -4;\ j = 7;\ k = 1;\ temp = 3\}]$
Iteration 8) $2 \rightarrow 3$ : $[3b,\ \{i = -4;\ j = 7;\ k = 1;\ temp = 3\}]$
Iteration 9) $3 \rightarrow 4$ : $[4b,\ \{i = -8;\ j = 9;\ k = 1;\ temp = 1\}]$
Iteration 10) $4 \rightarrow 5$ : $[5b,\ \{i = -8;\ j = 9;\ k = 1;\ temp = 1\}]$
Iteration 11) $4 \rightarrow 7$ : NOT REACHABLE
Iteration 12) $5 \rightarrow 6$ : $[6b,\ \{i = -8;\ j = 9;\ k = 2;\ temp = 1\}]$
...
Iteration 55) $6 \rightarrow 2$ : $[2z,\ \{i = -40;\ j = 25;\ k = 10;\ temp = -15\}]$
Iteration 56) $2 \rightarrow 3$ : NOT REACHABLE

Figure 2.5: Simulating the run for example in Figure 2.3

states are updated accordingly. For example, when an `EVEN` number and an `ODD` number is added, the result is set to `ODD`. At merge points, if the parities of the same variable is the same, then the parity value is kept for that variable. If the parities are different at join points, the abstract value is set to `UNKNOWN` after merging.

In Figure 2.6 and in Figure 2.7, we show lattices for two different abstract domains. The first figure shows the lattice for an integer variable that uses explicit value tracking which is similar to the explicit state tracking simulation that we have shown above. It assigns a variable to its integer values, or the 'TOP' element which is denoted by $\top$ or to the 'BOTTOM' value which is denoted by $\perp$. $\top$ element states that the value of an element is 'UNKNOWN' and it might have any value possible during the runtime of the program. $\perp$ states that this state is not reachable from the initial state of the program.

Figure 2.7 shows the lattice for the parity abstraction domain. So, this abstraction maps a variable to `ODD`, `EVEN`, $\top$ or $\perp$. For example, if a variable is assigned to 4, the abstraction function would map this variable to `EVEN` and if its parity cannot be decided, to $\top$.

An illustration of static analysis using abstract interpretation with parity domain and merging at join points is shown in Figure 2.8. Until iteration 7, it is straightforward to follow the steps of the analysis. With iteration 7, a new abstract state for `location 2` is produced. Since there was already another abstract state for this location which was produced after

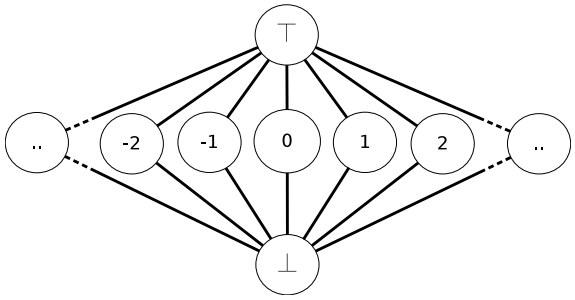


Figure 2.6: Lattice for explicit value tracking

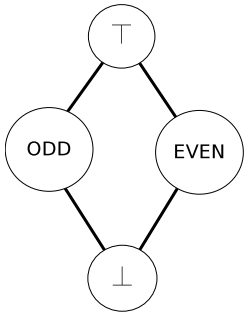


Figure 2.7: Lattice for parity domain

iteration 1, they are merged and a single abstract state for this location is updated. Note that since we represent the states for a single location with only one abstract element, we do not label locations with letters such as $2a$ and $2b$ with abstract interpretation. For the variable `k`, the parity was `EVEN` in the previously computed state and the new state is set to `ODD` for it. They are merged and the analysis cannot determine what might be the real value of the variable. Thus `k` is set to $\top$. `i` was `EVEN` and `j` was `ODD` in both of the merged states. So, even after merging, their abstract values remain same.

At iterations 5 and 15, an appropriate check on condition `temp == 0` can detect that we check for equality of an odd number to an even number and it can conclude that reaching to `location 7` from `location 4` is not possible. At iteration 16, a new abstract state is produced for `location 2`. This new state is already over-approximated by the abstract state that was associated with `location 2` and there are no new states to discover, so the analysis saturates and it can terminate by concluding that the program does not violate the given safety condition.

Comparing the simulation that tracked all values explicitly as shown in Figure 2.5 and the static analysis using parity abstraction as shown in Figure 2.8 illustrates the power of abstract interpretation: Using an appropriate abstraction can speed up the verification process significantly. In the explicit tracking case where the execution of the program is simulated in its operations' order, if the loop had more iterations (that might be infinite in some cases) it would take 6 times the number of iterations steps to terminate. In the parity based abstraction used as an example, it would still terminate in 18 steps and that is independent of the size of the loop.

Iteration 0) : Initial State $[1, \{\}]$
Iteration 1) : $1 \rightarrow 2$ $[2, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \mathsf{EVEN}\}]$
Iteration 2) : $2 \rightarrow 3$ $[3, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \mathsf{EVEN}\}]$
Iteration 3) : $3 \rightarrow 4$ $[4, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \mathsf{EVEN}, temp = \mathsf{ODD}\}]$
Iteration 4) : $4 \rightarrow 5$ $[5, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \mathsf{EVEN}, temp = \mathsf{ODD}\}]$
Iteration 5) : $4 \rightarrow 7$ NOT REACHABLE
Iteration 6) : $5 \rightarrow 6$ $[6, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \mathsf{ODD},\ temp = \mathsf{ODD}\}]$
Iteration 7) : $6 \rightarrow 2$ $[2, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \mathsf{ODD},\ temp = \mathsf{ODD}\}]$
Iteration 8) : MERGED $[2, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \top,\ temp = \mathsf{ODD}\}]$
Iteration 9) : $2 \rightarrow 3$ $[3, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \top,\ temp = \mathsf{ODD}\}]$
Iteration 10) : MERGED $[3, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \top,\ temp = \mathsf{ODD}\}]$
Iteration 11) : $3 \rightarrow 4$ $[4, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \top,\ temp = \mathsf{ODD}\}]$
Iteration 12) : MERGED $[4, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \top,\ temp = \mathsf{ODD}\}]$
Iteration 13) : $4 \rightarrow 5$ $[5, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \top,\ temp = \mathsf{ODD}\}]$
Iteration 14) : MERGED $[5, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \top,\ temp = \mathsf{ODD}\}]$
Iteration 15) : $4 \rightarrow 7$ NOT REACHABLE
Iteration 16) : $5 \rightarrow 6$ $[6, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \top,\ temp = \mathsf{ODD}\}]$
Iteration 17) : MERGED $[6, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \top,\ temp = \mathsf{ODD}\}]$
Iteration 18) : $6 \rightarrow 2$ $[2, \{i = \mathsf{EVEN},\ j = \mathsf{ODD},\ k = \top,\ temp = \mathsf{ODD}\}]$

Figure 2.8: Abstract interpretation based on parity for the example in Figure 2.3

### 2.2.3 Software Model Checking

Software model checking is the process of verifying software systems automatically using a model of the program [43]. A model of the program reflects the behaviour of the program and is constructed using the source code. Usually manually constructed invariants using user input such as assertions or annotations are used for model checking. The verification process is summarized in three steps [43]:

- **Modeling**: Modeling requires the construction of transition system that can be used by the verification system. A model of the system consists of states and transitions [59].

- **Specification**: The properties that must be satisfied should be specified. This can be done automatically or by using user annotations and there might be several ways of asserting a specification such as using automata, invariant languages or inserting assertions and error labels in the program.

- **Verification**: This step uses the model to generate states using the transition system. Most tools check for the satisfiability of the given specification on-the-fly by comparing

them to produced states. Since model checking may also produce false alarms, user interaction may be required at verification steps too. On the other hand, several methods have been proposed to tackle this problem which will be introduced in this chapter.

Model checking techniques can be classified into two categories [59]. Explicit model checking techniques use a similar approach to static analysis. A state transition graph is built by executing transitions on the model and encoding the states of the system using an appropriate formalism. Usually the states are encoded as logical formula. Temporal logic has been widely used since Clarke and Emerson introduced it as a model checking instrument [41]. In order to make explicit model checking faster, hashing and indexing techniques are used [78]. States are stored in a hash table so that comparison of states becomes easier [59]. Symbolic model checking techniques use data structures to represent a set of states instead of storing a separate state as a result of a transition. One of the most commonly used structures is Binary Decision Diagrams (BDDs) [35]. BDDs are used to represent boolean formulas using a boolean tree. They are more compact than disjunctive and conjunctive normal forms. Efficient techniques have been developed to eliminate redundant nodes in BDDs to make them smaller [43]. BDDs enable the model checking tools to compare states very fast and make it possible for automated tools to check for the equality of states which has fundamental importance for scalability of model checking algorithms.

### 2.2.4 Predicate Analysis

Predicate analysis is the most commonly used model checking technique for verification of software. It is similar to abstract interpretation techniques in a way that it uses abstraction, but it has a major difference because it does not have a pre-defined abstract domain but the abstract domain is built based on the program [59]. It is a model checking technique because the verification is run on a model of the program called the boolean program [11]. The concept of *boolean program* is implemented by the tool `Bebop` and it is a C program which has only boolean variables [13]. A boolean program is built by mapping concrete states of the program to abstract states which are boolean variables. This mapping is provided by a set of predicates. Predicate analysis was first introduced by Graf and Saidi but their technique was not automated. The tool SLAM introduced automation of the process through boolean programs [11, 63].

A predicate is a quantifier-free formula. Let us denote the set of predicates with $\mathcal{P}$. The boolean program is built based on the program statements and how they change the value of predicates in $\mathcal{P}$. A formula $\psi$ over predicates from $\mathcal{P}$ represent the region $r_\psi$ which has the concrete states that imply $\psi$: $r_\psi = \{c \mid c \models \psi\}$. The set of concrete states that are reachable from a set of concrete states following an operation $op \in Ops$ is defined by the strongest postcondition operator $\mathsf{SP}_{op}$. $\mathsf{SP}_{op}(\psi)$ computes the set of reachable states from $\psi$. Given the formula $\psi$ and a variable $x \in X$,

$$\mathsf{SP}_{op}(\psi) = \begin{cases} \exists x' : \psi_{[x \mapsto x']} \wedge (x = e_{[x \mapsto x']}) & \text{if } op \text{ is an assignment operation } x := e \\ \psi \wedge p & \text{if } op \text{ is an assume operation } assume(p) \end{cases}$$

where $[x \mapsto x']$ denotes the renaming of the formula or assignment operand obtained by replacing occurrences of $x$ by $x'$ [17].

The reachability problem for the predicate analysis becomes checking the feasibility of a path $\mathsf{SP}_\sigma(\psi) = \mathsf{SP}_{op_n}(\ldots \mathsf{SP}_{op_1}(\psi)\ldots)$ given a program path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n)\rangle$ where for every i with $1 \leq i \leq n$, we have $(l_{i-1}, op_i, l_i) \in G$. For a final program location $l_n$ and an initial predicate *true*, the set of reachable concrete states for the path $\sigma$ is shown by $(l_n, \mathsf{SP}_\sigma(true))$. If $\mathsf{SP}_\sigma(true)$ is satisfiable, the path $\sigma$ is feasible. Otherwise, it is an unreachable path.

$\pi \in \mathcal{P}$ is called a precision which is a subset of predicates of variables in $\mathcal{P}$. For efficiency, there is usually a mapping from locations to a precision, such that when computing a predicate abstraction of a state, instead of using all predicates in $\mathcal{P}$, a relevant set $\pi$ of predicates is used. We denote a formula computed using predicates from a precision $\pi$ by $\psi^\pi$. There are two different methods to compute the predicate abstraction of a concrete state, the first one is cartesian abstraction [12, 17]. Denoted by $\psi^\pi_{\mathbb{C}}$, it is computed by $\psi^\pi_{\mathbb{C}} = \bigwedge \{p \in \pi \mid \psi \Rightarrow p\}$. So, it is computed by the conjunction of predicates in $\pi$ that is entailed by $\psi$. The second method is boolean abstraction. The boolean abstraction of a formula is the strongest combination of predicates from $\pi$ that is entailed by the formula $\psi$ [12, 17, 88]. Denoted by $\psi^\pi_{\mathbb{B}}$, the boolean abstraction of a formula is computed as follows: For each predicate $p_i \in \pi$, a propositional variable $v_i$ is introduced and all satisfying assignments of $v_i$ for all $i$ in the formula $\psi \wedge \bigwedge_{p_i \in \pi}(p_i \Leftrightarrow v_i)$ is enumerated by the SMT

$\mathsf{a} \Leftrightarrow (j+i) \leq -3$
$\mathsf{b} \Leftrightarrow (j+i) == -1$
$\mathsf{c} \Leftrightarrow (j+i) == 1$
$\mathsf{d} \Leftrightarrow (j+i) == 3$
$\mathsf{e} \Leftrightarrow (j+i) == 5$
$\mathsf{f} \Leftrightarrow temp == 0$
$\mathsf{g} \Leftrightarrow i == 0$

Figure 2.9: List of predicates for the predicate analysis of the example program shown in Figure 2.3

solver. $f_{\pi_i}$ is the conjunction of all predicates that has their propositional variables $v_i$ which was set to *true*. The disjunction of all $f_\pi$ builds the boolean abstraction of the formula $\psi$.

### 2.2.5 Predicate Analysis Example

In this section there is an illustration of predicate analysis on the example given in Figure 2.3 and its CFA given in Figure 2.4. Assume that we have the set $\mathcal{P}$ that has predicates in Figure 2.9. For instance, predicate $(j+i) \leq -3$ is represented by $\mathsf{a}$, so in the abstract state, if $\mathsf{a}$ holds, the concrete state of the program satisfies $(j+i) \leq -3$.

In Figure 2.10, the process is shown. The first iteration sets the abstraction to *true* which implies that all variables may take any value. By using the cartesian abstraction approach explained above, abstractions are computed using predicates in $\mathcal{P}$. At iteration 4, the operation from `location 4` to `location 7` is an assume operation and we have $\neg\mathsf{f}$ in the abstract state formula $\psi$ for the abstraction computed for `location 4`. The condition statement is `temp == 0` and the conjunction of this condition with $\neg\mathsf{f}$ is unsatisfiable because $\neg\mathsf{f}$ implies that $temp \neq 0$. After iteration 7, only states that are newly computed are shown. For example after iteration 8, the reached states has two elements for `location 3` but we show only the new state for the sake of the simplicity. At iteration 27, a state is computed that was already computed before by iteration 21, so there are no remaining states to discover and the analysis terminates by reporting that the error condition is not reachable.

### 2.2.6 Abstract Reachability Tree

An abstract reachability tree (ART) is a representation of the reachable state space of a program. When the analysis is path-sensitive and flow-sensitive such as model checking or

Iteration 0) : Initial State $[1, \{true\}]$
Iteration 1) : $1 \rightarrow 2$ $[2a, \{e, \neg a, \neg b, \neg c, \neg d\}]$
Iteration 2) : $2 \rightarrow 3$ $[3a, \{e, \neg a, \neg b, \neg c, \neg d\}]$
Iteration 3) : $3 \rightarrow 4$ $[4a, \{d, \neg a, \neg b, \neg c, \neg e, \neg f\}]$
Iteration 4) : $4 \rightarrow 7$ $[7a, false]$
Iteration 5) : $4 \rightarrow 5$ $[5a, \{d, \neg a, \neg b, \neg c, \neg e, \neg f\}]$
Iteration 6) : $5 \rightarrow 6$ $[6a, \{d, \neg a, \neg b, \neg c, \neg e, \neg f\}]$
Iteration 7) : $6 \rightarrow 2$ $[2b, \{d, \neg a, \neg b, \neg c, \neg e, \neg f\}]$
Iteration 8) : $2 \rightarrow 3$ $[3b, \{d, \neg a, \neg b, \neg c, \neg e, \neg f\}]$
Iteration 9) : $3 \rightarrow 4$ $[4b, \{c, \neg a, \neg b, \neg d, \neg e, \neg f\}]$
Iteration 10) : $4 \rightarrow 7$ $[7b, false]$
Iteration 11) : $4 \rightarrow 5$ $[5b, \{c, \neg a, \neg b, \neg d, \neg e, \neg f\}]$
Iteration 12) : $5 \rightarrow 6$ $[6b, \{c, \neg a, \neg b, \neg d, \neg e, \neg f\}]$
Iteration 13) : $6 \rightarrow 2$ $[2c, \{c, \neg a, \neg b, \neg d, \neg e, \neg f\}]$
Iteration 14) : $2 \rightarrow 3$ $[3c, \{c, \neg a, \neg b, \neg d, \neg e, \neg f\}]$
Iteration 15) : $3 \rightarrow 4$ $[4c, \{b, \neg a, \neg c, \neg d, \neg e, \neg f\}]$
Iteration 16) : $4 \rightarrow 7$ $[7c, false]$
Iteration 17) : $4 \rightarrow 5$ $[5c, \{b, \neg a, \neg c, \neg d, \neg e, \neg f\}]$
Iteration 18) : $5 \rightarrow 6$ $[6c, \{b, \neg a, \neg c, \neg d, \neg e, \neg f\}]$
Iteration 19) : $6 \rightarrow 2$ $[2d, \{b, \neg a, \neg c, \neg d, \neg e, \neg f\}]$
Iteration 20) : $2 \rightarrow 3$ $[3d, \{b, \neg a, \neg c, \neg d, \neg e, \neg f\}]$
Iteration 21) : $3 \rightarrow 4$ $[4d, \{a, \neg b, \neg c, \neg d, \neg e, \neg f\}]$
Iteration 22) : $4 \rightarrow 7$ $[7d, false]$
Iteration 23) : $4 \rightarrow 5$ $[5d, \{a, \neg b, \neg c, \neg d, \neg e, \neg f\}]$
Iteration 24) : $5 \rightarrow 6$ $[6d, \{a, \neg b, \neg c, \neg d, \neg e, \neg f\}]$
Iteration 25) : $6 \rightarrow 2$ $[2e, \{a, \neg b, \neg c, \neg d, \neg e, \neg f\}]$
Iteration 26) : $2 \rightarrow 3$ $[3e, \{a, \neg b, \neg c, \neg d, \neg e, \neg f\}]$
Iteration 27) : $3 \rightarrow 4$ $[4e, \{a, \neg b, \neg c, \neg d, \neg e, \neg f\}]$

Figure 2.10: Predicate analysis process for the working example

predicate analysis, the abstract states are never merged and all possible execution paths of the program are explored separately. Therefore, the state space of the program can be modeled as an ART. An ART node is a reachable state of the program [18]. The root of the ART is the initial state and the edges correspond to the edges of the CFA. So if a node $n$ of the ART represents the state $c$ and its successor node $n'$ represents the state $c'$, then we have the $c \xrightarrow{g} c'$ relation and the edge between $n$ and $n'$ is labeled by $g$ on the ART. A *path* of the ART is a possible execution path of the program.

In Figure 2.11, we present an example of an abstract reachability tree. The ART in the figure is the state space for the predicate analysis described in Figure 2.10 for the

predicates in Figure 2.9 and the program in Figure 2.3. The edges of the ART are one-to-one representation of the iterations of the analysis as shown in Figure 2.10. The nodes contain the location for the abstract state computed. In order to distinguish between different states computed for the same location, we append a letter to each state. The locations were shown in the CFA of the program in Figure 2.4. Usually, tools assign a unique integer id for each node of the ART and the locations (or program counter) are a part of the abstract state. The analysis starts with the initial state which is the root of the ART. The dashed boxes present the abstract states computed with their associated nodes. For example, iteration 1 is from location 1 to location 2 and it represents the execution of the CFA edge `i = 0; j = 5; k = 0;`. As shown in Figure 2.10, the abstracted state for the location 2 is $(l = 2, \{e, \neg a, \neg b, \neg c, \neg d\})$. The location $l = 2$ is shown in the node as $2a$ and the state $\{e, \neg a, \neg b, \neg c, \neg d\}$ is shown next to the abstract node in the dashed box. If the bottom element, $\perp$ (*false*) is computed for a location, the abstract state is shown with a box which is filled with gray. In the example ART, `7a, 7b, 7c` and `7d` are computed to be *false* and they are unreachable. The last computed state, $4e$, with the gray font is covered by another element on the ART. They have the same location and same state, so the element `4e` is covered by the element `4d`. We show the coverage relation using a dashed line and we show an arrow from the covered state to the covering state. Abstract reachability graphs (ARG) are similar to abstract reachability trees with the difference that they may have some nodes with several incoming edges. Note that ARGs do not form loops as well, so the incoming edge is never from a child or grandchild of the node.

### 2.2.7 Counterexample-guided Abstraction Refinement

One of the main challenges in predicate analysis is the construction of predicates list. The solution to this problem is counterexample-guided abstraction refinement (CEGAR) [42,71]. This technique starts the analysis with a set of predicates (possibly no predicates) and whenever an error is found, its genuineness is checked. If the error is not genuine, the CEGAR algorithm uses the path to the error to refine the predicates. It generates predicates to track and they are added into $\mathcal{P}$. The analysis is restarted after each refinement until the analysis loop terminates or a real error is found.

Craig interpolants are used to construct predicates from proofs in Blast [18,68,71]. A Craig interpolant of a formula is computed as follows [56]: Given a pair formulas $\varphi^+$ and $\varphi^-$ such that $\varphi^+ \wedge \varphi^-$ is unsatisfiable, the craig interpolant for these formulas is $\psi$. $\psi$ satisfies

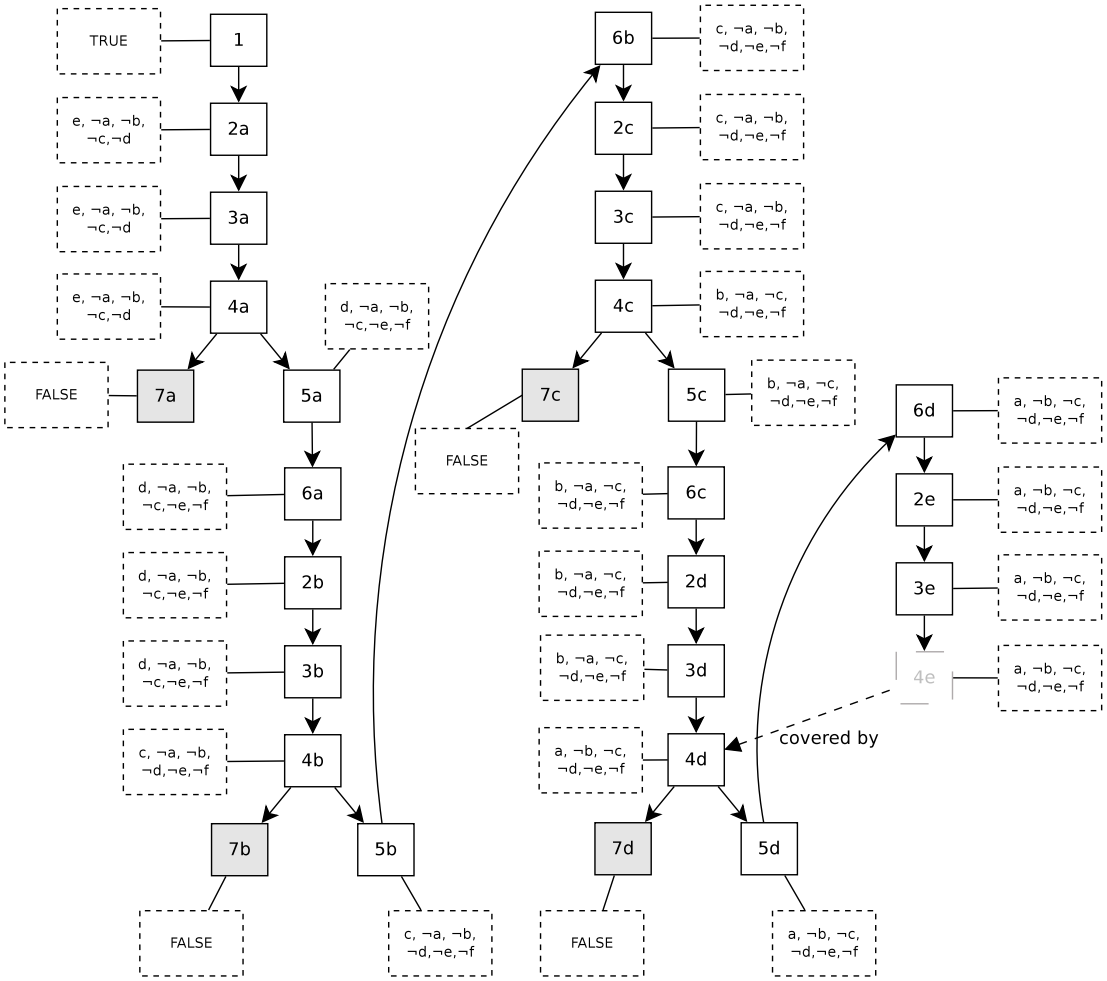


Figure 2.11: Example ART for the analysis explained in Figure 2.10

three properties: (i) $\varphi^- \Rightarrow \psi$, (ii) $\psi \wedge \varphi^+$ is unsatisfiable, and (iii) $\psi$ contains symbols that are only common in $\varphi^+$ and $\varphi^-$ [18].

The predicate generation approach intuitively works as follows: For each program location on the error path, a formula is built for the path until this location and another formula after this location. The Craig interpolant of these two locations gives us the predicate that needs to be added and tracked after this location. Suppose that we start with no predicates for predicate analysis of the example given in Figure 2.3. The following path that reaches the error location will be extracted:
$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 7$ and the following formula will be produced for the error path:

$$(i_1 = 0) \wedge (j_1 = 5) \wedge (k_1 = 0) \wedge (k_1 < 10) \wedge (i_2 = i_1 - 4) \wedge$$
$$(j_2 = j_1 + 2) \wedge (temp = i_2 + j_2) \wedge (temp = 0)$$

When computing the formula to represent the path, static single assignment (SSA) form is used which assigns a value to a variable only one time [18]. The decision procedure concludes that the formula is not satisfiable, so the error is not real. The first step is to compute the interpolant at node 2 for the CFA shown in Figure 2.4. So we compute the interpolant for

$\varphi^+ : (i_1 = 0) \wedge (j_1 = 5) \wedge (k_1 = 0)$
and
$\varphi^- : (k_1 < 10) \wedge (i_2 = i_1 - 4) \wedge (j_2 = j_1 + 2) \wedge (temp = i_2 + j_2) \wedge (temp = 0)$

In this example $\varphi^+$ represents the path to location 2 and $\varphi^-$ represents the remaining part of the path. An interpolant for $\varphi^-$ and $\varphi^+$ is $(i + j \geq 5)$ and this predicate is added to the list of predicates. This procedure is repeated for all program locations on the error path.

## 2.3 Configurable Program Analysis

Configurable program analysis was first introduced by Beyer et.al. [24]. CPA framework is a recent concept that aims to close the gap between static analysis and model checking of software.

As described in above sections, static analysis and model checking have different focuses on checking properties of programs. Although each of the techniques can be defined in terms of the other, there is considerable difference in practice [24,101,102]. Program analysis methods focus on extracting simple facts usually by running a path-insensitive analysis with merging at join points to reach a fixpoint. On the other hand, model checking algorithms explore the state space of the program by building the ART until there are no more elements to be discovered. That makes program analysis faster but less precise, thus letting it perform better on large-scale code and makes model checking more precise but less efficient.

Using a common formalism for different analysis techniques, CPA makes it possible to use intermediate settings that are configured for each analysis separately. This makes it possible for developers to adjust the analysis based on their programs and properties that they check for in order to increase precision and efficiency of the analysis.

In this section, we introduce the formalism of CPA and describe some analysis techniques that we use for our experiments in terms of their CPA descriptions.

### 2.3.1 Definition of CPA

A configurable program analysis is defined by the abstract domain of the analysis and operators that define the behaviour on join points and how to stop the analysis. Previous research proposed techniques for combining abstract interpretations based on combining domains of several analyses [49, 66]. These techniques provide mechanisms to represent the abstract domain using a unified lattice and describe a new operator to merge elements on join points.

The CPA framework, on the other hand, provides mechanisms to define how the analysis will proceed for the chosen analysis. Typical behaviour for static analysis at join points of the CFA is to merge the abstract element which is already computed for a location and the newly computed state. The merging computes an element which is an upper bound for merged elements, thus loosing precision. Model checkers do not merge elements at join points and continue with the analysis by computing new states for elements separately and they build the ART of the program. The behaviour of the analysis at the join points is defined by the *merge operator*. The *termination check* decides whether the analysis should continue exploring new elements on a path. Typically, static analysis stops when the element reaches to a fixpoint on a location which implies that there are no new states to be discovered for that location. Model checkers stop exploring a path when a newly computed stated is

covered by another element on the ART which implies that this state was already explored. The CPA framework provides mechanisms to combine abstract domains and to choose a suitable *merge operator* and a *termination check* for each analysis. The task of handling combinations is handled by the *composite merge operator* and the *composite termination check*. The *composite transfer relation* also provides flexibility to configure the analysis engine.

### 2.3.2 Configurable Program Analysis with Dynamic Precision Adjustment

Beyer et.al. introduced a complementary concept, dynamic precision adjustment for configurable program analysis [25]. While configuring the analysis engine helps the user to tune the efficiency and precision, abstract domains of analyses cannot be changed while running the analysis. Running several analyses in parallel by configuring them may be useful for tuning the domains of these analyses. That is where the dynamic precision can be used for.

The precision of an analysis defines how the abstraction will map the concrete states of a program to the abstract domain. For example, introducing more relevant predicates for the predicate analysis through refinement makes the process more precise. More facts and relations can be represented if there are more boolean variables to represent concrete states and the analysis will be more accurate. The dynamic precision approach lets the user to define the precision of the analysis and provides mechanisms to update the precision dynamically during the analysis. These mechanisms make it possible to change the precision of an analysis based on the results of other analyses that are executed in parallel. For instance, an analysis that explicitly tracks the values of variables and a predicate analysis can be combined using the CPA formalism. The analysis may start by tracking all possible values for all the variables explicitly. When a specified threshold which limits the number of values to be kept for a variable is hit, this variable can be removed from the list of tracked variables. Then, a predicate can be introduced to represent the relations of this variable [25]. Precisions of used analyses are determined by a *set of precisions* for each analysis. Precisions' dynamic adjustment is handled by a *precision adjustment function* which is called by the main reachability algorithm.

### 2.3.3 CPA Formalism

Although the framework for program analysis with dynamic precision is defined as CPA+ to distinguish it from the CPA framework, we will refer to configurable program analysis with dynamic precision simply as CPA for the rest of the thesis, because we always use precisions to define our analyses. In this section, we define the configurable program analysis with dynamic precision adjustment formally.

A configurable program analysis $\mathbb{C} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$, consists of 6 components [25]:

1. *The Abstract Domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$. The abstract domain D defines how the concrete states of the program will be represented in the abstract space. It basically determines the precision and efficiency of the analysis algorithm. An abstract domain $D$ consists of three components:

   - The set $C$ is the set of concrete states of the program as explained in Section 2.1.1.
   - The semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ consists of a set $E$ of abstract elements, the top element $\top \in E$ of the lattice, the bottom element $\bot \in E$ of the lattice, a binary relation $\sqsubseteq \subseteq E \times E$ that orders lattice elements, and a total function $\sqcup : E \times E \to E$ that is used as a join operator. The join operator $\sqcup$ computes the least upper bound for two elements. $\top$ is the least upper bound and $\bot$ is the greatest lower bound of $E$.
   - The concretization function $[\![\cdot]\!] : E \to 2^C$ maps an abstract element from the lattice to a set of concrete states which is represented by that abstract element.

2. *The set* $\Pi$ *of precisions* keeps possible precisions of the abstract elements. While the analysis continues, a precision from $\Pi$ is used for computing an abstract element by the analysis. A pair $(e, \pi)$ is called an abstract element $e$ with precision $\pi$ and the precision information defines the behaviour of the analysis on abstract element $e$.

3. *The transfer relation* $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ assigns a set of abstract states $e'$ with precision $p$ to an abstract element $e$. The set $e'$ of abstract states with precision $p$ is the set of abstract successors of $e$ computed using a CFA edge $g$ . If there is $(e, g, e, \pi) \in \rightsquigarrow$ then we write $e \overset{g}{\rightsquigarrow} (e', \pi)$. If there exists a $g$, to simply show that we have a successor relation between an abstract element and its set of successors, we write $e \rightsquigarrow (e', \pi)$.

4. *The merge operator* $\mathsf{merge} : E \times E \times \Pi \to E$ combines two abstract elements. The $\mathsf{merge}$ operator weakens the second element using the parameters of the operator which are the first abstract element and the precision. Note that the merge operator is not the same thing as the join operator $\sqcup$ of the lattice but $\mathsf{merge}$ can be configured based on the behaviour of the join operator. There are two $\mathsf{merge}$ operators that we commonly use: $\mathsf{merge}^{sep}(e, e', \pi) = e'$ which does not join the elements and returns the second element for a more precise analysis (that is the behaviour of traditional model checking) and $\mathsf{merge}^{join}(e, e', \pi) = e \sqcup e'$ which uses the join operator of the lattice (that is the behaviour of traditional static analysis).

5. *The termination check* $\mathsf{stop} : E \times 2^E \times \Pi \to \mathbb{B}$ checks whether an abstract element is covered by a set of abstract elements. $\mathsf{stop}$ takes three parameters and checks whether the first parameter with the given precision is a subset of a set of states given as the second parameter. For example, the set of states given as the second parameter might be the reached set of the analysis and the $\mathsf{stop}$ might check whether any element in the reached set subsumes the element given as the first parameter using the order relation $\sqsubseteq$ on the lattice. The $\mathsf{stop}$ operator does not have to behave same as $\sqsubseteq$ though. There are two $\mathsf{stop}$ operators that are commonly used: $\mathsf{stop}^{sep}(e, R, \pi) = (\exists e' \in R : e \sqsubseteq e')$ which returns `true` if there is any element in the reached set that is an upper bound of the checked element. $\mathsf{stop}^{join}(e, R, \pi) = (e \sqsubseteq \bigsqcup_{e' \in R} e')$ which returns `true` if the conjunction of all elements in the reached set is an upper bound for the checked element.

6. *The precision adjustment function* $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ computes a new abstract state with precision using an abstract state with precision and a set of abstract states with precisions. The $\mathsf{prec}$ operator may apply widening on the abstract state and may modify the precision as well. The widening operators are used to ensure that the analysis terminates by over-approximating the abstract state (loosing precision) after a number of iterations [51]. One advantage of $\mathsf{prec}$ operator is that it can be configured to act like a typical widening operator which is used to decrease the precision of the element or as a strengthening operator which can be used to increase the precision of the element. It does not have to be configured to act in a static way through the analysis, but rather its behaviour can be determined based on the abstract element and precisions [103].

The elements of a CPA have to fulfill the following requirements to satisfy the soundness of the algorithm [24, 25]:

(a) $[\![\top]\!] = C$ and $[\![\bot]\!] = \emptyset$
The top element represents the entire state space of the concrete states and the bottom element represents a state that does not correspond to a concrete state which is possible to be produced during the runtime of the program.

(b) $\forall e, e' \in E : e \sqsubseteq e' \Rightarrow [\![e]\!] \subseteq [\![e']\!]$
If an abstract element $e$ is lower than another abstract element $e'$ on the lattice, the set of concrete states which are represented by $e$ is a subset of the set of concrete states which are represented by $e'$.

(c) $\forall e, e' \in E : [\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$
The join operator produces a set of states which is a precise representation of joined states or it over-approximates the set of represented concrete states.

(d) $\forall e \in E, g \in G, \pi \in \Pi : \bigcup_{e \overset{g}{\rightsquigarrow} (e', \pi)} [\![e']\!] \supseteq \bigcup_{c \in [\![e]\!]} \{c' | c \overset{g}{\rightarrow} c'\}$
The transfer relation over-approximates an operation given a fixed precision.

(e) $\forall e, e' \in E, \pi \in \Pi : e' \sqsubseteq \mathsf{merge}(e, e', \pi)$
The $\mathsf{merge}$ operator's result can be equal to the second parameter or it can be an upper bound of the second parameter.

(f) $\forall e \in E, R \in E, \pi \in \Pi : \mathsf{stop}(e, R, \pi) = \mathit{true} \Rightarrow [\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!]$
If an abstract element $e$ is covered by a set $R$ of abstract states, concrete states represented by $e$ is a subset of some concrete states which is represented by the abstract states in $R$.

(g) $\forall e, e' \in E,\ p, p' \in \Pi, R \subseteq E \times \Pi : (e', p') = \mathsf{prec}(e, p, R) \Rightarrow [\![e]\!] \subseteq [\![e']\!]$
The result of the $\mathsf{prec}$ can be only more abstract than the first parameter.

More detailed information on the CPA framework and its soundness is given by Beyer et. al. [24, 25, 103].

### 2.3.4 CPA Algorithm

The CPA algorithm is a reachability algorithm which computes a new set of reachable abstract states with precision at each iteration. So, the algorithm computes a set of states which represents the set of an over-approximation of concrete states of the analyzed program. The algorithm presented in Algorithm 1 is taken from [25].

The CPA algorithm is started with three parameters. The first parameter is the configurable program analysis $\mathbb{C}$ which defines the components of the abstract state computation: The domain $D$, the precisions $\Pi$, the transfer relation $\rightsquigarrow$, the merge operator merge, the termination check stop and the precision adjustment function prec define the input CPA. The algorithm has two sets which keep abstract states with precisions and update them through the iterations of the algorithm: The set reached is the reached set which keeps abstract states with precisions that are computed during the analysis. The initial reached set is $R_0$. The set waitlist is the waitlist which keeps abstract states with precisions that are to be processed by the algorithm (i.e., keeps the states for which the abstract successors will be computed). The initial reached set is $W_0$. The algorithm runs until there are no elements left to be processed in the set waitlist.

The abstract state computation starts with an initial element and initial precision $e_0$ and $\pi_0$. At the beginning of each iteration, an abstract element $e$ and its precision $\pi$ is retrieved from the set waitlist. First, the abstract successors of the element $e$ is computed using $\rightsquigarrow$. Then each computed abstract successor and it precision is adjusted by prec using the set reached and $\pi$. Now, each adjusted element and precision is combined with abstract elements with precisions from the set reached by merge. If the merged state is different than the second parameter because the merge operation added more information to the abstract state, the second parameter (the abstract element from the set reached and its precision) is removed from the set waitlist and from the set reached and the merged element with its precision is added to both sets. At the end of the main loop's iteration, the newly computed abstract state and its precision is added to the set reached and the set waitlist if it is not detected to be covered by the existing elements in the reached set. The coverage is checked by the termination check stop. Every step of the algorithm which computes a reachable abstract element use successor computation, precision adjustment and merging operators. For all these steps, even when an abstract state is replaced, it is replaced by an abstract

element which represents a superset of the represented concrete states and they all preserve the soundness of the algorithm.

**Theorem 2.3.1** *(Soundness of CPA Algorithm). Given a CPA $\mathbb{C}$, a program $P$ and sets $R_0$ and $W_0$ of initial states, if the CPA algorithm given in Algorithm 1 terminates, it computes a sound over-approximation of the reachable concrete states of the program. Refer to Theorem 3.1 for details of the soundness of CPA algorithm and proofs in Théoduloz's thesis [103].*

**Algorithm 1** *CPA*$(\mathbb{C}, R_0, W_0)$

**Input:** a CPA $\mathbb{C} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
a set $R_0 \subseteq E \times \Pi$ of abstract states with precision,
a subset $W_0 \subseteq R_0$ of frontier abstract states with precision,
where $E$ denotes the set of elements of the semi-lattice of $D$
**Output:** a set of reachable abstract states with precision,
a subset of frontier abstract states with precision
**Variables:** two sets reached and waitlist of elements of $E \times \Pi$

```
reached := R_0;
waitlist := W_0;
while waitlist ≠ ∅ do
  choose (e, π) from waitlist; remove (e, π) from waitlist;
  for each ê with e⇝(ê, π) do
    // Adjust the precision.
    (e', π') := prec(ê, π, reached);
    for each (e'', π'') ∈ reached do
      // Combine with existing abstract state.
      e_new := merge(e', e'', π');
      if e_new ≠ e'' then
        waitlist := (waitlist ∪ {(e_new, π')}) \ {(e'', π'')};
        reached := (reached ∪ {(e_new, π')}) \ {(e'', π'')};
    // Add new abstract state?
    if ¬ stop(e', {e | (e, ·) ∈ reached}, π') then
      waitlist := waitlist ∪ {(e', π')};
      reached := reached ∪ {(e', π')};
return (reached, ∅)
```

## 2.4 Related Work: Automated Software Analysis Tools

We classified automated software analysis tools into two categories: First, tools running an analysis based on the abstract interpretation of the program which we refer as static

analysis tools; second, model checking tools that check if a specification holds by exploring all possible states of the program on a given model of the software [59, 60]. Static analysis tools aim to extract simple facts about the program efficiently (i.e., using less time and resources) but they are not very precise when computing the abstraction of a program. Therefore, static analysis is more efficient in detecting bugs but not very useful in proving the absence of errors based on a given specification. On the other hand, model checking tools build the state space of the program more precisely and are also useful in showing that the program satisfies a given specification. In comparison to static analysis tools, they spend more time and resources and do not scale to industrial-size code. The two methods can be formalized by common operators. The difference in theory is not significant anymore, but in practice it still exists due to the different focus of their verification purposes [24].

Static analysis tools can be classified based on the domains that they use because the abstract domain determines the behaviour of the analysis. Numerical abstract domains extract facts about numerical variables of the program [55, 90]. The octagon abstract domain can represent relations between variables in the form ($ax + by \leq c$) [91]. The octagon domain represents the relations of variables on difference bound matrices. The operators for the analysis modify these matrices to proceed with the analysis [90]. If there is a large number of variables, the computation might be expensive but octagon domain is efficient to represent simple relations between programs. Octahedra domain and convex polyhedra domain extends the octagon domain to represent relations among more program variables. Octahedra domain uses a decision diagram with nodes consisting of variables and leaves consisting of integer variables instead of matrices to represent a domain element. Its operations are optimized to modify this decision diagram [40]. Convex polyhedra domain provides the most precise numerical abstract domain with ability to express precise constraints [55, 67]. *Parma Polyhedra Library* (PPL) collects many numerical domains under a library and provides operations to use them to run a fully automated software analysis [4]. ASTRÉE uses several abstract domains to run abstract interpretation-based static analysis [52]. It has many optimizations to scale to the analysis of large-size programs such as keeping localized matrices for octagon domain instead of propagating the entire matrix while the analysis progresses [52, 54]. ASTRÉE also provides a framework that enables different domains to communicate and letting the tool to switch to a more precise or efficient domain on-the-fly [53].

Another type of static analysis aims to extract facts about the data structures of the program. Points-to analysis and alias analysis are commonly used techniques to check the behaviour of pointers. Points-to analysis tries to determine which pointer accesses to which memory location and alias analysis tries to determine pointers that point to the same memory location [73]. There have been many techniques proposed for pointer analysis as well which is classified based on their efficiency and precision. Shape analysis refers to a more common problem which includes analysis which infers facts about data structures of the program [100]. Shape analysis models the heap by shape graphs which consists of heap nodes. It also has the power to express the type of the data structure such as a linked list or a tree. Beyer et al. recently introduced lazy shape analysis which stores shape graphs locally and uses a refinement procedure similar to CEGAR approach for predicate analysis [23]. The refinement detects and adds field predicates and pointer variables to be tracked locally. By running an explicit heap analysis and selecting a relevant shape structure from a domain of structures, they also can discover the class of the shape [26]. Shape analysis with refinement is implemented in Blast.

Model checking techniques can be used to prove the safety of a program for a given specification. One of their main advantages is their capability to produce a counterexample that shows the user how an error condition can be reached [59]. Predicate analysis is one of the commonly used tools for software model checking due to its success to refine the abstraction domain through refinement process [8,18]. SLAM analyzes the programs using a counterexample driven refinement combining different tools for computing the abstraction [8, 11, 48]. SLAM uses C2bp to build the boolean program of the analyzed software. C2bp converts the C program to a boolean program based on the given predicates [12]. Then, BEBOP uses the boolean program to symbolically check if an error state is satisfiable [13,14]. NEWTON checks the feasibility of an error path and generates predicates by symbolically executing the path program [15]. SLAM has been successfully used for checking specifications on Windows device drivers by eliminating many false alarms [6]. A new version of SLAM uses a coarser-grained abstraction that summarizes a few statements into one ART node and computes the abstraction. It is also improved by optimized predicate generation algorithm which uses information from the progress of the analysis.

Blast runs predicate analysis that uses a lazy abstraction [18, 71]. Lazy abstraction discovers predicates locally and uses only relevant predicates for a location to avoid unnecessary work when computing the abstraction. Blast has been used for many practical

software engineering problems. For example, it has been been shown that it might be a useful tool for extreme programming, by incrementally checking properties of an incomplete program [70]. It is also used for generating test cases using the counterexamples tracked by the tool and generating safety proofs that acts like a certificate for systems code [16, 69].

Blast uses a refinement technique based on Craig interpolants [18]. Various approaches have been proposed to make the refinement more precise and efficient. One of the main drawbacks of typical CEGAR approach used by model checkers like Blast or SLAM is handling of code that requires universal quantifiers to represent invariants. In the presence of a loop, for example, the CEGAR algorithm may unroll the loop over and over to add a new predicate to the precision. A technique called 'Path Invariants' treats an error path like a program and such a program is called a path program [22]. An invariant for this path program is generated using an invariant generation approach that can handle a number of combined theories including uninterpreted functions and universal quantifiers over arrays [21]. Such an approach removes many infeasible paths for the sequence rather than eliminating only one. Another approach that was implemented using Blast to make the CEGAR loop efficient is 'Path Slicing' [82]. This approach detects and removes operations on a path that does not contribute to the infeasibility of a path. It works by running a dataflow analysis on the path which is usually an efficient analysis and if it removes irrelevant parts that are in loops, it might significantly speed up the analysis.

SATabs uses a symbolic approach for analyzing software [46]. Tools that take the approach of SLAM and Blast make an exponential number of calls to theorem provers to compute the boolean program. They use theorem provers such as Simplify or Zapato [9,58]. SATabs exploits the recent advances in SAT solvers and makes a single call to a SAT solver instead of making an exponential number of calls [45]. The representation for an operation of the program is constructed using symbolic execution. Khurshid et al. translate the program to another language that can be symbolically executed by any model checker [84]. They used a constraint solver tool Korat for transformation and JPF for symbolic execution. That work aims to handle programs with complex data structures. JPF is an explicit state model checking tool for Java programs and supports handling of nondeterminism by annotations and uses heuristics for guiding the search [64].

Calysto conducts a symbolic search on generated verification conditions [2, 3]. It is an extended static analyzer which uses maximally-shared graphs instead of BDDs for efficiency [3]. Maximally-shared graphs represent the flow of the program and the analysis

tools operates on them using BDD-like optimizations. Extended static checking tools are descendants of LINT [83]. They focus on efficiency by sacrificing soundness and completeness to detect bugs. ESC/JAVA is a extended static checking tool for Java programs that requires user annotations to compute the validity of postconditions using preconditions and program statements [61].

Bounded model checking (BMC) is another imprecise approach that was inspired from hardware verification [30]. With the BMC framework, loops are unrolled a given number of times and if there are bugs in the program that requires more unrolling for the loop, they are missed [59]. The BMC approach runs an 'abstraction, model checking, refinement' loop similar to BLAST and SLAM but instead of using BDDs to represent states, it takes a similar approach as SATABS and passes the symbolic formula to a SAT solver. CBMC is a popular tool that implements BMC approach [44]. SATURN also implements BMC to scale to large programs [107]. SATURN unrolls the loops two times hoping that most bugs manifest themselves in that range. It models many variables including pointers and heap data.

In this thesis, we present a software analysis tool CPACHECKER that tries to close the gap between static analysis and model checking by providing interfaces for common theoretical operators. It includes several analyses including an optimized predicate analysis that runs a CEGAR-based refinement algorithm to adjust predicates and for predicate abstraction.

# Chapter 3

# Implementation of CPA Framework: CPAchecker

As discussed before, the main aim of software analysis research is to make the process as precise and fast as possible. There have been many practical approaches developed to achieve this. The basic difficulty of combining or comparing these techniques is that they are usually built on different tools using different formalisms. Parsing the source code with different parser front-ends, deploying different theorem provers for abstraction or formula construction tasks, or reporting the results in different formats make it particularly difficult to use different tools together.

In Chapter 2, we have discussed that whether the analysis is based on model checking or based on lattice-based static analysis, it can be formalized as a recently emerged formalism called CPA [27]. Providing a framework for building the CFA of the program and an algorithm that serves as the execution engine, CPA framework can be used to combine different tools within the same analysis framework. We implemented a tool, CPAchecker, for analysis of C programs that provides a set of interfaces for easy integration of any analysis which can be defined as a CPA [27].

CPAchecker is an open source software released under the Apache 2.0 license. It is online at `http://cpachecker.sosy-lab.org` and can be downloaded and used for analysis of C code with the existing CPAs or for inserting and configuring a user-defined analysis as a CPA.

## 3.1 CPAs for the Analyses Used in the Thesis

In this section, we define the CPAs of analyses used in this thesis. We define the CPA for the location analysis, for an explicit value tracking analysis and for predicate analysis.

### 3.1.1 CPA for Location Analysis

The CPA for location analysis is used for running the standard reachability algorithm based on the reachability of a location. It keeps track of the state of the program counter explicitly and is used for composition for all of our CPAs since our verification problems are based on the reachability of an error label on program's CFA [24, 25]. $\mathbb{L} = (D_{\mathbb{L}}, \Pi_{\mathbb{L}}, \rightsquigarrow_{\mathbb{L}}, \mathsf{merge}_{\mathbb{L}}, \mathsf{stop}_{\mathbb{L}}, \mathsf{prec}_{\mathbb{L}})$ specifies the CPA for location analysis and it is defined as follows:

1. The domain $D_{\mathbb{L}}$ is based on the flat lattice for the set $L$ of program locations: $D_{\mathbb{L}} = (C, \mathcal{E}_{\mathbb{L}}, [\![\cdot]\!])$, with $\mathcal{E}_{\mathbb{L}} = ((L \cup \{\top\}), \sqsubseteq)$, $l \sqsubseteq l'$ if $l = l'$ or $l' = \top$, $[\![\top]\!] = C$, and for all $l$ in $L$, $[\![l]\!] = \{c \in C \mid c(pc) = l\}$.

2. The set $\Pi_{\mathbb{L}}$ of precisions contains only one precision $\pi_{\mathbb{L}}$.

3. The transfer relation $\rightsquigarrow_{\mathbb{L}}$ has the transfer $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} (l', \pi)$ if $g = (l, \cdot, l')$.

4. The merge operator does not combine elements: $\mathsf{merge}_{\mathbb{L}}(l, l', \pi) = l'$

5. The termination check returns true if the current element is already in the reached set: $\mathsf{stop}_{\mathbb{L}}(l, R, \pi) = (l \in R)$.

6. The precision is never changed: $\mathsf{prec}_{\mathbb{L}}(l, \pi, R) = (l, \pi)$.

### 3.1.2 CPA for Explicit Analysis

We define an explicit analysis which has a similar behaviour to constant propagation [36]. Explicit analysis is used to keep track of explicit values of integer variables through the execution of the program. Our implementation of explicit analysis can be configured but for this thesis, we only use the explicit analysis with $\mathsf{merge}^{sep}$ and with $\pi = \infty$, so we will define merge operator and precision based on our usage. The definition of CPA for explicit analysis $\mathbb{E} = (D_{\mathbb{E}}, \Pi_{\mathbb{E}}, \rightsquigarrow_{\mathbb{E}}, \mathsf{merge}_{\mathbb{E}}, \mathsf{stop}_{\mathbb{E}}, \mathsf{prec}_{\mathbb{E}})$ is repeated from [25]:

1. The abstract domain $D_{\mathbb{E}} = (C, \mathcal{E}, [\![\cdot]\!])$ consists of the following components: $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ is the semi-lattice where $E$ is the set of the integer assignments to variables. The top element $\top$ represents the state which no variables is assigned a value and the bottom element $\bot$ denotes the state which an assignment is not possible (a state that is not reachable from the initial state of the program).

2. The set of precisions $\Pi_{\mathbb{E}} = \{\pi \mid \pi : X \to (\mathbb{N} \cup \{\infty\})\}$ where a precision $\pi \in \Pi_{\mathbb{E}}$ defines a threshold for the number of values to be assigned to a variable. $\pi(x) = \infty$ states that all values will be kept for the variable $x$. We set $\infty$ as the precision for our use of explicit analysis so the remaining definitions will be based on $\pi(x) = \infty$ [25].

3. The transfer relation $\rightsquigarrow_{\mathbb{E}}$ has the transfer $e \overset{g}{\rightsquigarrow} (e', \pi)$ if

   (a) $\forall x \in X$ and $g = (\cdot, assume(p), \cdot)$ we have

   $$e'(x) = \begin{cases} \top, & \text{if } [p/e] \Rightarrow (x = \top) \\ \bot, & \text{if } [p/e] \Rightarrow \mathit{false} \text{ or } \exists y \in X : e(y) = \bot \\ c, & \text{if } [p/e] \Rightarrow (x = c) \\ e(x), & \text{otherwise} \end{cases}$$

   where $[p/e]$ denotes a predicate with all occurences of a variable $x \in X$ is replaced by $e(x)$. So, if the state $e$ does not satisfy the condition for the assume edge, bottom element $\bot$ is produced. If the element satisfies the condition, then each assigned value is transferred to the successor element $e'$ for each variable.

   (b) $\forall x \in X$ and $g = (\cdot, y := exp, \cdot)$ we have

   $$e'(x) = \begin{cases} [exp/e], & \text{if } x = y \\ e(x), & \text{otherwise} \end{cases}$$

   where $[exp/e]$ represents the evaluation of an expression over $X$ for the assignments of abstract element $e$ in the following way:

$$[exp/e] = \begin{cases} \bot, & \text{if } x \in X \text{ occurs in } exp \text{ with } e(x) = \bot \\ \top, & \text{if } x \in X \text{ occurs in } exp \text{ with } e(x) = \top \\ & \quad \text{otherwise where } exp \text{ evaluates to } c \\ c, & \quad \text{for each occurence of } x \in X, \\ & \quad x \text{ is replaced by } e(x) \end{cases}$$

So, if an assignment expression assigns an integer value to a variable, the assingment of the variable in $e'$ is updated accordingly. If the assingment is a more complex operation such as addition of two variables, then the expression is evaluated based on their assigned values from the element $e$ and the value of the operation is assigned to the variable in $e'$ accordingly.

4. The merge operator does not merge elements on join locations, so we use $\mathsf{merge}^{sep}$: $\mathsf{merge}_{\mathbb{E}}(e, e', \pi) = e$.

5. The termination check compares states individually, so we use $\mathsf{stop}^{sep}$: $\mathsf{stop}_{\mathbb{E}}(e, R, \pi) = (\exists e' \in R : e \sqsubseteq e')$.

6. The precision adjustment function does not change the element and the precision: $\mathsf{prec}_{\mathbb{E}}(e, \pi, R) = (e, \pi)$ since we use it with a constant precision $\pi(x) = \infty$.

### 3.1.3 CPA for Adjustable-block Encoding

In this section, we define the CPA of a recent approach we developed called adjustable-block encoding for predicate analysis (ABE) [17,28,106]. Predicate abstraction is a highly precise software model checking technique as explained in Chapter 2. The predicate abstraction of a concrete state of the program is computed using SMT solvers. Usually, the bottleneck for predicate analysis is the expensive calls made to the SMT solvers. Research focuses on making predicate abstraction more efficient as well as preserving its precision. One way to solve the problem is minimizing the number of calls to the SMT solvers.

With recent developments in SMT solvers, tools such as MathSAT can handle even very large-size formulas efficiently nowadays [34, 38]. We introduced a technique called large-block encoding (LBE) for predicate analysis that utilizes the power of SMT solvers [17]. LBE transforms the CFA of the program into another graph where larger parts of the CFA are encoded as a single node. This transformation groups loop-free parts of the CFA into

a single node. The sequential statements are modeled by building the conjunction of the statements and the branches are modeled by building disjunctions of different branches explained [17]. LBE approach is developed on CPAchecker and the experiments have shown that it dramatically improves the analysis compared to traditional predicate analysis which we define as single-block encoding (SBE) [17].

LBE approach is a very efficient method for utilizing the power of SMT solvers and it can be defined in a more flexible way in the CPA framework. When the CFA is transformed into another graph, we cannot use it in combination with other CPAs in a flexible way anymore. Also, the LBE approach groups nodes using a pre-defined mechanism and the grouping can be made more flexible. We introduced a more flexible encoding called adjustable-block encoding for predicate analysis [28]. In ABE, CFA is not transformed into another graph and computing the formula for a group of nodes is handled on-the-fly. The predicate analysis with ABE has two phases on its transfer relation: (i) Updating the formula that represents a group of nodes (a step that basically builds the formula syntactically to represent a path) and, (ii) computing the abstraction for a concrete state (a step that makes a call to SMT solvers). An `abstraction location` is the location where the actual abstraction is computed by making a call to the SMT solver. In ABE, the abstraction locations can be given to the analysis as a parameter. For example, the user may wish to compute the abstraction after 10 CFA edges have been processed. This property can be specified as the parameter and every $10^{th}$ location on the ARG will be used as an `abstraction location` [28,106]. An `abstraction element` is a predicate analysis element computed for abstraction locations.

The formula that represents the operations between two abstraction locations is called the *disjunctive path formula* and it is denoted by $\varphi$. The abstraction computed for a state is called the *abstraction formula* and it is denoted by $\psi$ [28]. An abstract state contains both formulas, $\varphi$ and $\psi$: $\psi$ denoting the abstraction for the last state on the path to the current state that an abstraction is computed for and $\varphi$ denoting the disjunctive path formula from that last abstraction state to the current node. Intuitively, the *disjunctive path formula* is computed as follows: The assignment operations are added to the formula by computing the conjunction of the current formula and the formula that models the assignment. If there is a conditional statement, the disjunctive path formulas for both branches are computed separately and locally and at their join point, locally computed formulas are merged by their disjunction. So, contrary to the typical model checking approach for predicate analysis

(SBE), we merge at some join points when we use ABE. Merging operator has to satisfy some requirements to meet the soundness criteria and to preserve precision: (i) Only non-abstraction locations are merged, so the operator is used to compute the *disjunctive path formula*, but never used to merge abstraction formulas. (ii) Two nodes with the same location but with different abstraction formulas $\varphi_1$ and $\varphi_2$ such that $\varphi_1 \neq \varphi_2$, are never merged even when they are non-abstraction locations. (iii) Two nodes are merged if their abstraction formulas are computed at the same location node such that we can still run a model checking algorithm instead of an algorithm similar to static analysis [106].

**CPA for ABE.** We formalized ABE as a CPA and implemented it in our tool CPAchecker. The ABE approach provides a intermediate setting between the classical approach to predicate analysis which is defined as SBE and a more symbolic predicate analysis which is defined as LBE. SBE computes the abstraction for every node and LBE computes the abstraction only on loop-heads and entry nodes of functions. Both SBE and LBE were defined as CPAs [17,24]. For ABE, we need an operator which we call the block-adjustment operator and we denote that operator by the function $\mathsf{blk}$. $\mathsf{blk}$ takes two arguments, an abstraction $e$ and a CFA edge $g$. If $\mathsf{blk}(e, g)$ for a given CFA edge $g = (l, op, l')$ returns *true*, a new abstraction formula $\psi'$ is computed and the disjunctive path formula is reset to *true*. If $\mathsf{blk}(e, g)$ returns false, the abstraction formula is not updated and a new disjunctive path formula $\varphi'$ is computed using the edge $g$. It is the operator $\mathsf{blk}$ that enables to use the predicate analysis with intermediate settings. The user can define the $\mathsf{blk}$ operator's behaviour based on the length of an ARG path or based on the properties of a CFA location. Note that $\mathsf{blk}$ always returns *true* for error locations (locations for error labels) in our analysis.

The CPA for predicate analysis with ABE $\mathbb{P} = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \mathsf{merge}_{\mathbb{P}}, \mathsf{stop}_{\mathbb{P}}, \mathsf{prec}_{\mathbb{P}})$ is defined in [28] as follows:

1. The abstract domain $D_{\mathbb{P}} = (C, \mathcal{E}, [\![\cdot]\!])$ consists of the set $C$ of concrete state, the semi-lattice $\mathcal{E}$ and the concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{E} = (\mathcal{E}, \top, \bot, \sqsubseteq, \sqcup)$ defines the domain of the analysis:

   (a) The lattice elements (abstract states) $e \in E$ consist of four elements $(l, \psi, l^{\psi}, \varphi) \in L \times \mathcal{P} \times L \times \mathcal{P}$ where $L$ is the list of CFA locations plus the top element for the location CPA, and $\mathcal{P}$ is a set of quantifier-free predicates over variables from $X$. $l$ models the location of the abstract state, $\psi$ is the abstraction formula that is a combination of predicates from the precision $\pi \in \Pi_{\mathbb{P}} : 2^{\mathcal{P}}$ for the element $e$, $l^{\psi}$ is

the location that $\psi$ was computed for and $\varphi$ is the disjunctive path formula for the path from $l$ to $l^{\psi}$. For an abstraction element, $l^{\psi}$ would be $l$ and the disjunctive path formula would be *true*.

(b) The top element $\top$ is the abstract state $(l_{\top}, \mathit{true}, l_{\top}, \mathit{true})$ which represents a state where there is no information about the state.

(c) The bottom element $\bot$ is the abstract state $(l_{\bot}, \mathit{false}, l_{\bot}, \mathit{false})$ which represents an unreachable state in the program. For a state to be unreachable, either $\varphi$ or $\psi$ should be *false*.

(d) The partial order $\sqsubseteq \subseteq E \times E$ is defined using abstraction and non-abstraction locations. Let $e_1 = (l_1, \psi_1, l^{\psi_1}, \varphi_1)$ and $e_2 = (l_2, \psi_2, l^{\psi_2}, \varphi_2)$:

$$e_1 \sqsubseteq e2 \iff \begin{cases} e_2 = \top \text{ or} & \\ (\psi_1 \Rightarrow \psi_2) & \text{where } (l_1 = l_2 = l^{\psi_1} = l^{\psi_2}) \text{ or} \\ (\varphi_1 \Rightarrow \varphi_2) & \text{where } ((l_1 = l_2) \wedge (l^{\psi_1} = l^{\psi_2}) \wedge (\psi_1 = \psi_2)) \end{cases}$$

The first condition is straightforward, if the second element is the top element it covers the first element. The second conditions indicates that, if both elements are abstraction elements on the same location, the partial order checks for the implication of abstraction formulas. The last condition indicates that, if the elements are on the same non-abstraction locations, and their abstraction formulas are the same, the partial order checks for the implication relation between disjunctive path formulas.

(e) The join operator $\sqcup : E \times E \to E$ returns the upper bound of two elements on the lattice.

The concretization function $[\![\cdot]\!] : E \to C$ maps the abstract states to the concrete states that they represent.

2. The set of precisions $\Pi_{\mathbb{P}} = 2^{\mathcal{P}}$ has the predicates for abstract states. If an analysis has the predicate $p$ in a precision $\pi$, then $p$ is used to compute the abstraction for the abstract elements with precision $\pi$. If $p$ is in the abstract state $e$, then $p$ is *true* for all concrete states represented by the abstract state $e$.

3. The transfer relation $\rightsquigarrow_{\mathbb{P}}$ has the transfer $e \overset{g}{\rightsquigarrow} (e', \pi)$ for an operation where $g$ is the edge from $l$ to $l'$, $g = (l,\ op,\ l')$, $e = (l, \psi, l^{\psi}, \varphi)$ and $e' = (l', \psi', l^{\psi'}, \varphi')$ if

$$\begin{cases} (\varphi' = \mathit{true}) \wedge (l' = l^{\psi'}) \wedge (\psi' = (\mathsf{SP}_{op}(\psi \wedge \varphi)^{\pi})) & \text{if } \mathsf{blk}(e, g) \\ (\varphi' = (\mathsf{SP}_{op}(\varphi))) \wedge (\psi' = \psi) \wedge (l^{\psi'} = l^{\psi}) & \text{otherwise} \end{cases}$$

The transfer relation is determined by the result of the $\mathsf{blk}$ function for a location. If the location is an abstraction location, the transfer relation updates the abstraction formula for the element $e'$ by computing the strongest postcondition of the conjunction of $\psi$ and $\varphi$ over the predicates from $\pi$. The disjunctive path formula, $\varphi'$ is reset to *true* and the abstraction location for the element is set to this location $l^{\psi'} = l'$. If the location is not an abstraction location, then only the disjunctive path formula, $\varphi'$ is updated using the operation from $g$.

4. The merge operator $\mathsf{merge}_{\mathbb{P}} : E \times E \times \Pi \to E$ takes two abstract states $e_1 = (l_1, \psi_1, l^{\psi_1}, \varphi_1)$ and $e_2 = (l_2, \psi_2, l^{\psi_2}, \varphi_2)$, and a precision $\pi$ and returns the merged of the two elements in the following way:

$$\mathsf{merge}_{\mathbb{P}}(e_1, e_2, \pi) = \begin{cases} (l_2, \psi_2, l^{\psi_2}, \varphi_1 \vee \varphi_2) & \text{if } (l_1 = l_2) \wedge (\psi_1 = \psi_2) \wedge (l^{\psi_1} = l^{\psi_2}) \\ e_2 & \text{otherwise} \end{cases}$$

We use a $\mathsf{merge}$ that acts like a $\mathsf{merge}^{join}$ for non-abstraction locations, and $\mathsf{merge}^{sep}$ for abstraction locations. If the location is a non-abstraction location and both elements have the same locations, the same abstraction locations and the same abstraction formulas, then two disjunctive path formulas are merged by computing their disjunction. Otherwise, the merge operator returns the second element.

5. The stop operator $\mathsf{stop}_{\mathbb{P}} : E \times 2^E \times \Pi \to \mathbb{B}$ compares the states individually using the partial order relation: $\forall e \in E, R \subseteq E : \mathit{stop}(e, R, \pi) = \exists e' \in R : (e \sqsubseteq e')$.

6. The precision adjustment function does not change the element and the precision: $\mathsf{prec}_{\mathbb{P}}(e, \pi, R) = (e, \pi)$

Note that the CPA for predicate analysis can be configured to act like a predicate analysis with SBE by returning *true* for every edge $g$ in the program and every abstract state $e \in E$

such that $\forall e, g : \mathsf{blk}(e, g) = \mathit{true}$. We can also use the CPA as a predicate analysis with LBE by returning *true* when g is an edge from a loop-head or from the entry node of a function. ABE and LBE obviously outperform SBE because of the significantly lower number of calls to the SMT solver. The experimental results comparing LBE, SBE and ABE are presented in our previous works [17,28]. ABE might introduce some advantages over LBE in addition. It would be possible to construct smaller formulas in case a program builds huge formulas for loop-free parts of the program and the formula becomes difficult for the SMT solver to solve. It also allows the user to define blk function in a flexible way, such as returning true when the size of the formula exceeds a threshold.

For most of our benchmark programs, LBE performs reasonably well and we will configure predicate analysis to use LBE approach for the rest of this thesis. The abstraction method is configurable for predicate analysis with ABE in CPAchecker, so the user might select between cartesian and boolean abstractions. We use boolean abstraction to compute the predicate abstraction for LBE.

## 3.2 Architecture and Implementation

Figure 3.1 shows the overview of the CPAchecker architecture which is taken from [27]. The analysis is run using the CFA of the source code. For the source code to be converted to its CFA representation, the C code should be pre-processed and reduced to an intermediate language called CIL [97]. The CIL reduction simplifies complex instructions of the code to simpler ones and produces a representation of the code which has the same behaviour with assume operations, assignment operations, function calls and function returns. CPAchecker uses the Eclipse plugin `CDT` which has a C parser. [1]

The CPA algorithm runs the main CPA algorithm given in Algorithm 1. The CPAs to be used by the algorithm is given as a parameter to the algorithm by the user. We also provide accesses to tools that can be used by any procedure or CPA during the analysis. We use JavaBDD[2] for BDD storage manipulation, MathSAT[3] and CSIsat[4] as the interpolation

[1] Available at http://www.eclipse.org/cdt.

[2] Available at http://javabdd.sourceforge.net.

[3] Available at http://mathsat4.disi.unitn.it.

[4] Available at http://www.sosy-lab.org/ dbeyer/CSIsat.

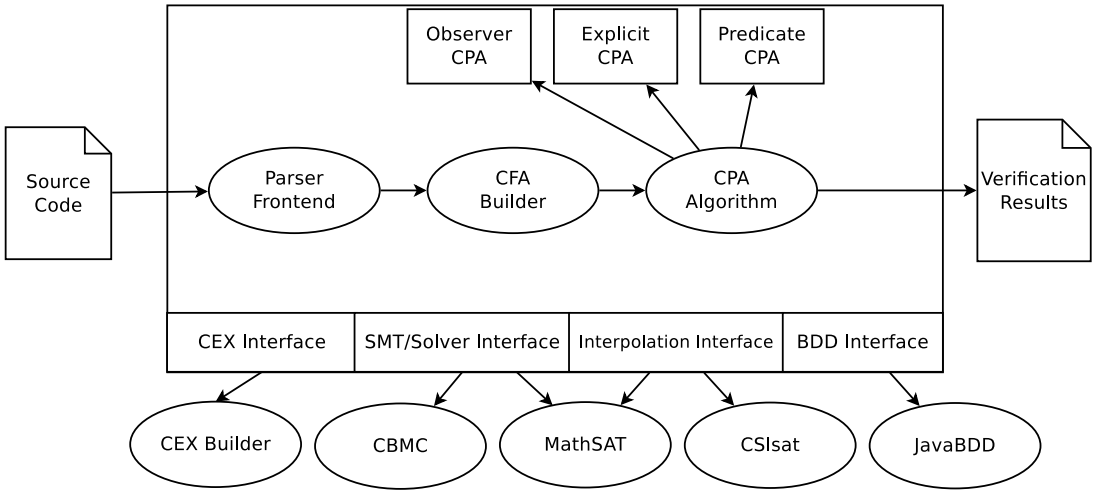


Figure 3.1: CPAchecker — Architecture overview taken from [27]

tools, MathSAT and CBMC[5] for the theorem solver or SMT interface and we provide a counter-example (CEX) builder to generate a C program from possible execution paths of the program. The analysis terminates with a verification result possibly giving information about the reachability of the given error condition, and summarize the work done, such as reporting the reachable states and statistics of the analysis.

The core of the analysis algorithm is the CPA algorithm. CPAchecker provides interfaces to implement the CPA framework explained in Chapter 2. Figure 3.2 gives an overview of the CPA algorithm implementation which is also from [27]. The CPA algorithm uses two data structures to run the algorithm: The CFA of the program and the CPA defined by the user. The CPA might be a *Composite CPA* or a single CPA which we call as a *Leaf CPA*. The *Composite CPA* is a composition of several *Leaf CPAs*. The top-right box in Figure 3.2 shows the interfaces required to implement a CPA. The bottom-right box shows the implementation of a *Leaf CPA* which implements all components and operations required. For example, if a CPA for explicit analysis is to be implemented, an *Explicit Analysis CPA* is built by providing implementations for CPA interfaces. Then, the *Explicit Analysis CPA* can be used to analyze a program by itself or can be combined with other CPAs such as *Predicate Abstraction CPA* [27]. Several different implementation for operators merge or stop can be provided and the user may decide which implementation to use when starting

[5]Available at http://www.cprover.org/cbmc.

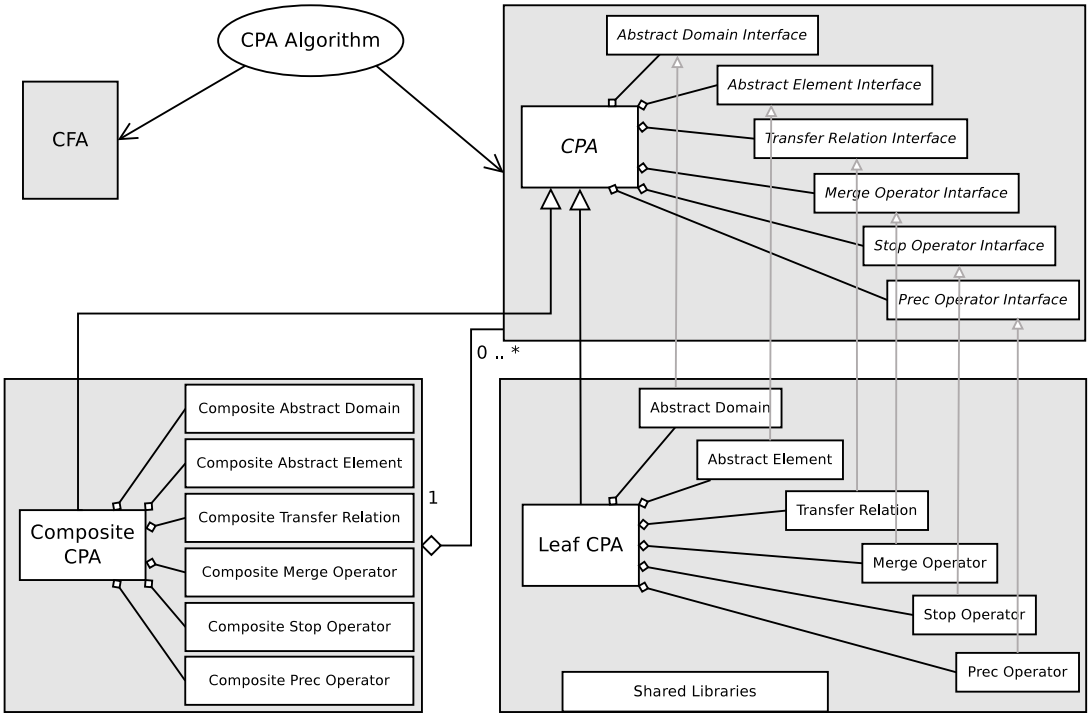


Figure 3.2: Interactions between CPAchecker components taken from [27]

the analysis. For the *Explicit Analysis CPA* example, the user may choose $\mathsf{merge}^{sep}$ as the merge operator to run a more precise path-sensitive and flow-sensitive analysis by building the ARG of the program. Alternatively, the user might pick $\mathsf{merge}^{join}$ as the merge operator to run a fast but less precise analysis which acts more like a static analysis. It is even possible to change the merge operator on-the-fly as some condition is met or provide a merge operator that is configured by the user providing that the operators satisfy the soundness conditions of CPA formalism.

Building a *Composite CPA* is fairly easy with CPAchecker. Any implemented CPA can be composed with any other CPA and it can be configured. Since they have to implement the CPA interfaces, no user interaction is required to use them together. Only when some special operations are required by two or more CPAs, the user must provide the implementation to handle it. For example, if we want to use *Explicit Analysis CPA* to generate some predicates to be used by the *Predicate Abstraction CPA*, then we need to implement this using precisions and the precision adjustment functions of these two CPAs.

## 3.3 CPAchecker Extensions

We extended CPAchecker to provide many techniques that are commonly used in practical software analysis. CPAchecker has also been used by researchers to implement their techniques [1, 37, 77]. In CPAchecker we provided the following CPAs and tools that can be used for analyzing software:

- An *explicit analysis CPA* that keeps track of the integer variables explicitly with different merge and stop operators.
- A *predicate abstraction CPA* that uses ABE and can be configured with several parameters.
- An *uninitialized variables CPA* that detects the usages of variables that have not been initialized.
- A *types CPA* to keep track of types of variables which is usually useful when combined with other CPAs.

- An *automaton CPA* makes it possible to guide the analysis using a given automaton. It also makes it possible to check for more complex properties than simple assertions or reachability of error labels in the code.
- An *ART CPA* which is used in combination with CPAs that need to build an ARG. This CPA is particularly useful when the analysis needs to run a refinement algorithm. After an infeasible error path is found, this path is constructed using the ARG, it is removed from the reached set and the analysis is restarted again. We provide an environment that can be easily used by developers to implement an analysis which has a refinement procedure using the ART CPA.
- We provide CBMC integration, so bit-precise bounded model checking tool CBMC can be called from CPAchecker when needed. We use CBMC to verify feasibility of error paths.
- A counterexample check algorithm with interpolation generation is implemented, so it can be called from any CPA if needed.
- CPAchecker has also an Eclipse plugin that can be used for software analysis while developing C code in Eclipse [105].

### 3.3.1 Algorithm Interface and CBMC Feasibility Check

If an external analysis tool, such as CBMC is to be integrated to CPAchecker, we need a more flexible main analysis algorithm. On the other hand, we do not want to change the configurable and flexible CPA algorithm by hard-coding extensions into it. So, we introduced an algorithm interface that is also implemented by the CPA algorithm. Therefore, if we want to use the CPA algorithm, we can call it as an algorithm and simply start the analysis. If we want to use an external tool, such as CBMC, the CPA algorithm will not be useful because it strictly requires proper CPA components and operations. Using the algorithm interface, we can also call CBMC as an external analysis tool and use its results.

In Figure 3.3, we show the architectural view of a custom algorithm that uses two different algorithms. The algorithm interface requires to implement methods to return the analysis result by returning sets reached and waitlist at the end of the analysis. For an external call, we can simply return waitlist $= \emptyset$ and insert an error element in the set reached

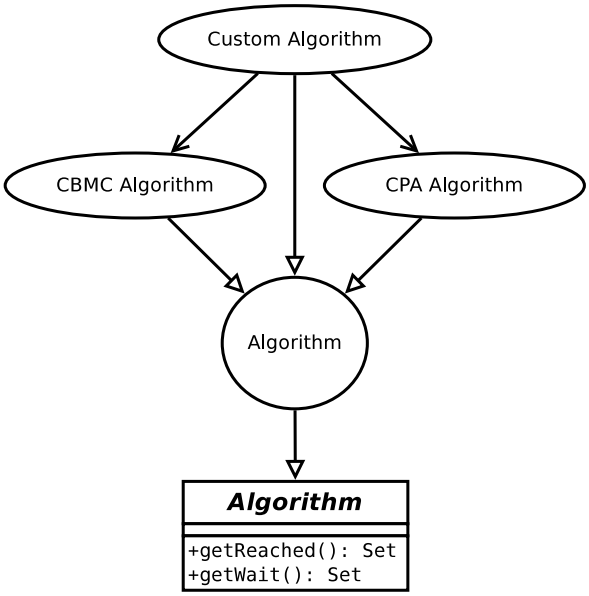


Figure 3.3: The algorithm interface is used by several algorithms

to indicate that the result of the analysis reached to an error condition. Or, we can insert a dummy element to waitlist and return to indicate that the analysis result is not sound.

Figure 3.3 shows an example algorithm that we implemented in CPAchecker using CBMC. We call CBMC as an external tool from CPAchecker and use its results to refine CPAchecker results [99]. CBMC runs a bit-precise bounded model checking algorithm. If the code contains loops with many iterations, CBMC usually requires a bound on the number of unwindings of the loops and the result of the CBMC is then imprecise. But, if we know that the program contains no loops or any other type or re-entrant code, CBMC is usually very effective. Another advantage of CBMC is its capability to run a bit-precise analysis. It can handle bitwise operations or can detect integer overflows unlike predicate analysis based model checkers like Blast or CPAchecker. The lack of support for operations and conditions that require an analysis at bit level, CPAchecker may produce false alarms. So, when the analysis ends, if an error label is reached, we build a program that corresponds to the error path of the program. That is possible by creating a C program and starting from the root of the ARG, adding each edge's operation on the error path to that C program as a line. Statements are added with no modifications and assume operations are added

as `_CPROVER_assume()` statements that CBMC can handle as assume statements. Such a program is called an error path program. Since the error path on ARG contains no loops, the generated program is free of loops and CBMC usually runs very fast on the error path programs. We use this algorithm to build a program that represents the error path and verify the feasibility of the path. If the error is due to some constructs that CPAchecker cannot handle, then this might be a false positive and we rule out this path and continue with the analysis if the error is determined to be on an unreachable path.

**Algorithm 2** *AnalysisWithFeasibilityCheck*$(A_1, A_2, \mathbb{C}, R_0, W_0)$

**Input:** a CBMC algorithm $A_1$
a CPA algorithm $A_2$
a CPA $\mathbb{C} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ with the *assumptions storage CPA*,
a set $R_0 \subseteq E \times \Pi$ of abstract states with precision,
a subset $W_0 \subseteq R_0$ of frontier abstract states with precision,
where $E$ denotes the set of elements of the semi-lattice of $D$
**Output:** a set of reachable abstract states with precision,
assumption generated for the analysis
**Variables:** two sets reached and waitlist, a boolean variable isfeasible to check the result from CBMC

```
reached := R_0;
waitlist := W_0;
while waitlist ≠ ∅ do
  (reached, waitlist) := A_2.run(reached, waitlist);
  if reached contains ERROR element e_e then
    isfeasible := A_1.run(reached)
    if isfeasible then
      return (reached, waitlist)
    else
      reached := reached \ e_e
return (reached, waitlist)
```

In Algorithm 2, we give an overview of the algorithm for feasibility checking with the CBMC algorithm. This algorithm corresponds to the *Custom Algorithm* shown in Figure 3.3. It takes two algorithms, a CBMC algorithm $A_1$ to call CBMC as an external tool and a CPA algorithm $A_2$ that takes typical CPA algorithm inputs. The algorithm runs the CPA algorithm just as a regular CPA. Then, when the analysis terminates, the result is checked and if there is an error element $e_e$ in reached, the CBMC algorithm is called with reached as input. Then, using the procedure explained above, the CBMC algorithm builds the error

path program for the error path and runs its model checking algorithm on it. If the error path is feasible, the analysis outputs the result as UNSAFE. If the path is detected to be infeasible, the error element $e_e$ is removed from reached and if there are any elements left in waitlist, the CPA algorithm is called again with waitlist and the remaining elements from reached. The analysis continues until no elements are left in waitlist or an error which is verified by the CBMC algorithm is found.

## 3.4 Experiments with CPAchecker

In this section, we introduce benchmark programs we used for CPAchecker experiments and explain details of the hardware architecture that is used to monitor the performance of the tool on benchmark programs. We also present the structure of tables that present experiments in the thesis.

### 3.4.1 Benchmark Programs and Configurations

In CPAchecker repository, there are a number of different types of benchmark programs that we use to test our algorithms. The first group of programs is referred as `locks` and shown with `test_locks`* in experiments tables. `locks` benchmark programs are artificially created to simulate a lock-release paradigm of a system. These programs acquires the lock and modifies a variable after that. At the end of the program, it is checked whether the lock and the variable adhere to the specification. These benchmark programs are designed to have many conditional statements, thus they have many branches in their ARGs. For typical model checking algorithms, it is difficult to verify these type of programs due to very large state space generated by many conditional statements. These programs were used in several projects before for performance experiments [17, 28].

The second group of programs is referred as `ntdrivers` benchmark and this benchmark includes programs `cdaudio`*, `floppy`*, `diskperf`*, and `kbfilter`*. These programs are part of driver software for Windows NT operating system. They are the simplified and modified versions of the original programs that are prepared to be analyzed by verification tools. These programs' basic behaviour is to make many function calls. They were used for comparison and performance experiments in several projects [17, 18, 24, 28].

The third group of programs is referred as `ssh` benchmark and include programs `ssh_clnt`*, and `ssh_srvr`* which are part of ssh protocol's client and server codes, respectively. They

are also simplified and modified versions of the original programs that are prepared to be analyzed by verification tools. These programs' structure is challenging for verification, since the main large block of the code is run in a loop and this large block has many conditional statements. The resulting ARG becomes huge due to many branches created by the assume edges and the loop. Also, `ssh` programs heavily use integer variables, make assignments to these variables and use relations between these integer variables to proceed which makes it suitable for a sound analysis. They were used for comparison and performance experiments in several projects [17, 18, 24, 28].

The forth group of programs is referred as `systemc` benchmark and include programs `kundu`*, `pc_sfifo`*, `pipeline`*, `token_ring`*, `transmitter`*, `bist_cell`*, `mem_slave_tlm`*, and `toy`* programs. These programs are part of the SystemC, a commonly used system design language [79, 86, 93]. The benchmark programs are converted to C programs and simplified and modified by Cimatti et al. [39]. We used benchmark programs from their repository [6]. The general structure of these programs consists of many function calls and code blocks with many conditional statements. Some of them also have loops and they usually require to build a huge ARG to check the safety of the programs. Note that the `toy`* programs in this benchmark set are not created by us but taken from the repository and they are designed to be challenging for model checkers by making many function calls with many branchings in all of its functions.

The simplification for these programs include removing accesses to heap and pre-processing the codes by Cil. Programs' size is from 200 lines to 3000 lines. In all programs, error labels `ERROR` are inserted, and we define the safety violation as the reachability of these labels. In other words, if none of the `ERROR` labels are reachable in a program, it is reported to be SAFE. Different versions of the same program are indicated using a different number for the program. The version might be different because the `ERROR` labels might be inserted to different sections of the code, or some parts of the code might have been altered to produce a different structure. We have two types of programs for all benchmark sets: (i) Programs with a known bug, and their names contain `BUG`. (ii) Programs that are known to be SAFE, and they do not contain `BUG` in their names. It should be noted that when we report a program to be UNSAFE or to be SAFE, we only refer to the reachability of `ERROR` labels

[6] `https://es.fbk.eu/people/roveri/tests/fmcad2010`

in the code. Thus, when we report SAFE, we do not claim the program to be free of any problems except the reachability of the error labels.

We show the result of the analysis by ✓if the analysis was successful. This indicates that if the program name contains `BUG`, the buggy state is reached; otherwise, the state space of the program is fully discovered and no error states were in the reached set for a program with no known bugs. If the analysis fails to report a result, or terminates by an imprecise result, it are indicated by '-'. Different cases for failure is explained for each table in the thesis if necessary. The performance results are presented in 'seconds' and they show the consumed processor time of the analysis tool. We use different time-limits for different experiments and we will specify the time-limit explicitly for each configuration for all the tables.

All experiments were performed on a machine with a 3.2 GHz Quad Core CPU and 16 GB of RAM. The operating system was Ubuntu 10.4 (64 bit), using Linux 2.6.32 as kernel and OpenJDK 1.6 as Java virtual machine unless otherwise stated. We used the CPAchecker components from revision 4080 of the repository with some modifications to the original version and CBMC in version 4.0 for running the experiments. All performance results that report the time spent are rounded to 3 significant digits.

### 3.4.2 Experiments for Comparison with Other Tools

In Table 3.1, we present results for predicate analysis with LBE and explicit analysis and compare it with results for the benchmark programs analyzed with CBMC, SATabs and Blast. For all examples, we limited the memory to 14 GB and 900 s. For explicit analysis, the Java heap was limited to 12 GB and for LBE to 4 GB to allow the remaining available memory to be used by the SMT and interpolation solvers.

The first columns lists the name of the program analyzed. The second column lists the results for CBMC where no fixed loop bound is used. We run CBMC 4.0 with options (`--error-label "ERROR" --32`). The third column lists the results for CBMC with a fixed loop bound 10, we run CBMC with options (`--error-label "ERROR" --32 --unwind 10`). The forth column lists the results for SATabs. (`--error-label "ERROR" --32`) options are used for SATabs 3.0 experiments. SATabs is used with `Cadence SMV` as the model checker [45, 46]. (`--32`) option for CBMC and SATabs specify that the integer variables are 32-bit integers in the analyzed programs. The other tools implicitly assume that programs use 32-bit integers, so we set the integer size to 32 bits for these analyzers as well.

Table 3.1: Comparions with other tools

| | CBMC | | | | SATABS | | BLAST | | Explicit | | Predicate | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $k = \infty$ | | $k = 10$ | | | | | | | | LBE | |
| `test_locks_14.BUG` | - | 813 | ✓ | 0.37 | ✓ | 0.22 | ✓ | 0.01 | ✓ | 289 | ✓ | 1.72 |
| `test_locks_15.BUG` | - | 801 | - | 0.42 | ✓ | 0.30 | ✓ | 0.01 | - | 543 | ✓ | 1.77 |
| `test_locks_5` | - | 342 | - | 0.07 | ✓ | 0.18 | ✓ | 0.80 | ✓ | 1.52 | ✓ | 1.62 |
| `test_locks_6` | - | 313 | - | 0.09 | ✓ | 0.29 | ✓ | 1.54 | ✓ | 2.08 | ✓ | 1.35 |
| `test_locks_7` | - | 343 | - | 0.12 | ✓ | 0.49 | ✓ | 2.92 | ✓ | 1.92 | ✓ | 1.61 |
| `test_locks_8` | - | 323 | - | 0.14 | ✓ | 0.73 | ✓ | 5.83 | ✓ | 2.82 | ✓ | 1.66 |
| `test_locks_9` | - | 340 | - | 0.16 | ✓ | 1.17 | ✓ | 11.3 | ✓ | 3.35 | ✓ | 1.41 |
| `test_locks_10` | - | 303 | - | 0.21 | ✓ | 1.52 | ✓ | 23.8 | ✓ | 6.28 | ✓ | 1.41 |
| `test_locks_11` | - | 371 | - | 0.24 | ✓ | 2.58 | ✓ | 52.6 | ✓ | 25.0 | ✓ | 1.42 |
| `test_locks_12` | - | 364 | - | 0.27 | ✓ | 3.76 | ✓ | 130 | ✓ | 122 | ✓ | 1.43 |
| `test_locks_13` | - | 339 | - | 0.31 | ✓ | 5.72 | ✓ | 370 | ✓ | 556 | ✓ | 1.44 |
| `test_locks_14` | - | 302 | - | 0.35 | ✓ | 7.61 | - | 900 | - | 900 | ✓ | 1.65 |
| `test_locks_15` | - | 365 | - | 0.40 | ✓ | 8.12 | - | 900 | - | 900 | ✓ | 1.45 |
| **Locks total** | - | 5320 | 1 | 3.15 | 13 | 32.7 | 11 | 2400 | 10 | 3350 | 13 | 19.9 |
| `cdaudio_simpl1_BUG` | ✓ | 0.99 | ✓ | 1.09 | - | 900 | ✓ | 40.7 | ✓ | 2.99 | ✓ | 9.49 |
| `floppy_simpl3_BUG` | ✓ | 0.16 | ✓ | 0.25 | ✓ | 139 | ✓ | 1.73 | ✓ | 2.47 | ✓ | 6.81 |
| `floppy_simpl4_BUG` | ✓ | 0.30 | ✓ | 0.40 | ✓ | 389 | ✓ | 1.79 | ✓ | 2.58 | ✓ | 10.2 |
| `kbfiltr_simpl2_BUG` | ✓ | 0.11 | ✓ | 0.21 | ✓ | 36.0 | ✓ | 3.88 | ✓ | 2.15 | ✓ | 4.00 |
| `cdaudio_simpl1` | ✓ | 1.01 | ✓ | 1.13 | - | 900 | ✓ | 81.2 | ✓ | 3.67 | ✓ | 17.2 |
| `diskperf_simpl1` | - | 140 | - | 0.35 | - | 900 | ✓ | 57.5 | - | 900 | ✓ | 15.1 |
| `floppy_simpl3` | ✓ | 0.16 | ✓ | 0.25 | ✓ | 468 | ✓ | 45.1 | ✓ | 2.62 | ✓ | 8.66 |
| `floppy_simpl4` | ✓ | 0.29 | ✓ | 0.38 | - | 900 | ✓ | 74.8 | ✓ | 3.32 | ✓ | 12.5 |
| `kbfiltr_simpl1` | ✓ | 0.05 | ✓ | 0.14 | ✓ | 11.8 | ✓ | 4.49 | ✓ | 2.39 | ✓ | 3.09 |
| `kbfiltr_simpl2` | ✓ | 0.11 | ✓ | 0.20 | ✓ | 29.3 | ✓ | 6.97 | ✓ | 2.89 | ✓ | 4.57 |
| **NT drivers total** | 9 | 143 | 9 | 4.40 | 6 | 4670 | 10 | 318 | 9 | 925 | 10 | 91.6 |
| `s3_clnt_1_BUG` | - | 281 | - | 4.25 | ✓ | 6.10 | ✓ | 5.39 | ✓ | 4.17 | ✓ | 3.11 |
| `s3_clnt_2_BUG` | - | 283 | - | 4.38 | ✓ | 5.94 | ✓ | 5.93 | ✓ | 4.15 | ✓ | 2.82 |
| `s3_clnt_3_BUG` | - | 281 | ✓ | 5.34 | ✓ | 6.69 | ✓ | 5.84 | ✓ | 3.99 | ✓ | 3.21 |
| `s3_clnt_4_BUG` | - | 283 | - | 4.38 | ✓ | 6.65 | ✓ | 5.41 | ✓ | 3.77 | ✓ | 3.65 |
| `s3_srvr_11_BUG` | - | 323 | ✓ | 6.73 | ✓ | 342 | - | 0.05 | ✓ | 2.57 | ✓ | 3.60 |
| `s3_srvr_12_BUG` | - | 365 | ✓ | 8.62 | ✓ | 554 | - | 0.04 | ✓ | 2.24 | ✓ | 5.62 |
| `s3_srvr_14_BUG` | - | 330 | - | 6.35 | ✓ | 13.0 | - | 0.04 | ✓ | 2.60 | ✓ | 2.48 |
| `s3_srvr_1_BUG` | - | 324 | ✓ | 6.55 | ✓ | 8.54 | ✓ | 22.6 | ✓ | 1.83 | ✓ | 2.76 |
| `s3_srvr_2_BUG` | - | 319 | ✓ | 6.29 | ✓ | 6.98 | ✓ | 67.7 | ✓ | 2.09 | ✓ | 2.85 |
| `s3_srvr_6_BUG` | - | 363 | ✓ | 7.08 | ✓ | 1.02 | ✓ | 168 | ✓ | 274 | ✓ | 1.63 |
| `s3_clnt_1` | - | 282 | - | 4.36 | ✓ | 29.4 | ✓ | 113 | ✓ | 8.07 | ✓ | 7.62 |
| `s3_clnt_2` | - | 283 | - | 4.65 | ✓ | 206 | ✓ | 149 | ✓ | 7.86 | ✓ | 6.72 |
| `s3_clnt_3` | - | 281 | - | 5.38 | ✓ | 145 | - | 99.5 | ✓ | 8.99 | ✓ | 5.82 |
| `s3_clnt_4` | - | 282 | - | 4.81 | ✓ | 149 | ✓ | 87.0 | ✓ | 8.29 | ✓ | 10.4 |
| `s3_srvr_1` | - | 338 | - | 4.15 | ✓ | 79.9 | - | 0.02 | ✓ | 2.58 | ✓ | 21.6 |
| `s3_srvr_1a` | - | 897 | - | 0.23 | ✓ | 0.50 | ✓ | 4.35 | - | 2.91 | ✓ | 2.82 |

Continued on next page

**Table 3.1 – continued from previous page**

| | CBMC $k=\infty$ | | CBMC $k=10$ | | SATAbs | | Blast | | Explicit | | Predicate LBE | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| s3_srvr_1b | - | 505 | - | 0.15 | ✓ | 0.16 | ✓ | 0.48 | - | 1.84 | ✓ | 1.71 |
| s3_srvr_2 | - | 318 | - | 5.58 | ✓ | 104 | - | 47.9 | ✓ | 2.87 | ✓ | 149 |
| s3_srvr_3 | - | 315 | - | 5.68 | ✓ | 75.1 | - | 36.0 | - | 2.73 | ✓ | 9.49 |
| s3_srvr_4 | - | 317 | - | 5.64 | ✓ | 103 | - | 56.7 | - | 2.70 | ✓ | 28.6 |
| s3_srvr_6 | - | 362 | - | 6.64 | ✓ | 611 | ✓ | 162 | ✓ | 241 | ✓ | 228 |
| s3_srvr_7 | - | 316 | - | 6.37 | ✓ | 585 | - | 19.1 | ✓ | 180 | ✓ | 46.2 |
| s3_srvr_8 | - | 337 | - | 6.13 | ✓ | 515 | ✓ | 47.8 | ✓ | 2.52 | ✓ | 21.7 |
| **SSH total** | - | 7990 | 6 | 120 | 23 | 3550 | 14 | 1100 | 19 | 774 | 23 | 571 |
| kundu1_BUG | - | 900 | - | 18.0 | ✓ | 15.9 | ✓ | 30.9 | ✓ | 2.24 | ✓ | 15.1 |
| kundu2_BUG | - | 900 | - | 127 | ✓ | 30.2 | - | 5.85 | ✓ | 1.89 | ✓ | 829 |
| pc_sfifo_1_BUG | - | 900 | ✓ | 13.1 | ✓ | 0.34 | ✓ | 0.50 | - | 900 | ✓ | 1.43 |
| pc_sfifo_2_BUG | - | 900 | ✓ | 7.11 | ✓ | 0.47 | ✓ | 0.51 | - | 900 | ✓ | 1.46 |
| pipeline_BUG | - | 900 | - | 900 | ✓ | 340 | - | 900 | - | 11.0 | - | 900 |
| token_ring.01.BUG | - | 103 | - | 22.0 | ✓ | 2.71 | - | 2.83 | - | 1.95 | ✓ | 3.72 |
| token_ring.02.BUG | - | 112 | - | 82.0 | ✓ | 15.4 | ✓ | 167 | - | 2.06 | ✓ | 32.9 |
| token_ring.03.BUG | - | 121 | - | 263 | ✓ | 36.4 | - | 12.5 | - | 2.60 | - | 900 |
| token_ring.04.BUG | - | 125 | - | 653 | ✓ | 83.0 | - | 6.33 | - | 2.90 | - | 900 |
| token_ring.05.BUG | - | 130 | - | 900 | ✓ | 230 | - | 25.3 | - | 3.81 | - | 900 |
| token_ring.06.BUG | - | 124 | - | 900 | ✓ | 569 | - | 8.86 | - | 4.94 | - | 900 |
| token_ring.07.BUG | - | 130 | - | 900 | - | 900 | - | 91.3 | - | 8.83 | - | 900 |
| token_ring.08.BUG | - | 127 | - | 900 | - | 900 | - | 11.0 | - | 29.1 | - | 900 |
| token_ring.09.BUG | - | 132 | - | 900 | - | 900 | - | 140 | - | 125 | - | 900 |
| token_ring.10.BUG | - | 128 | - | 900 | - | 900 | - | 14.0 | - | 501 | - | 900 |
| token_ring.11.BUG | - | 126 | - | 900 | - | 900 | - | 243 | - | 900 | - | 900 |
| token_ring.12.BUG | - | 132 | - | 900 | - | 900 | - | 17.4 | - | 900 | - | 900 |
| token_ring.13.BUG | - | 148 | - | 900 | - | 900 | - | 378 | - | 900 | - | 900 |
| token_ring.14.BUG | - | 131 | - | 900 | - | 900 | - | 17.7 | ✓ | 498 | - | 900 |
| token_ring.15.BUG | - | 133 | - | 900 | - | 900 | - | 378 | - | 900 | - | 900 |
| toy1_BUG | - | 900 | - | 34.7 | ✓ | 7.12 | ✓ | 812 | - | 2.78 | ✓ | 345 |
| toy2_BUG | - | 900 | - | 31.2 | ✓ | 5.85 | ✓ | 900 | ✓ | 2.21 | ✓ | 122 |
| transmitter.01.BUG | - | 125 | ✓ | 5.29 | ✓ | 0.93 | ✓ | 6.30 | ✓ | 1.55 | ✓ | 2.03 |
| transmitter.02.BUG | - | 137 | ✓ | 22.4 | ✓ | 9.48 | ✓ | 53.1 | ✓ | 1.73 | ✓ | 4.89 |
| transmitter.03.BUG | - | 127 | - | 94.1 | ✓ | 21.2 | ✓ | 213 | ✓ | 1.97 | ✓ | 34.5 |
| transmitter.04.BUG | - | 132 | - | 269 | ✓ | 39.3 | - | 900 | ✓ | 2.26 | - | 900 |
| transmitter.05.BUG | - | 133 | ✓ | 673 | ✓ | 73.0 | - | 900 | ✓ | 2.46 | - | 900 |
| transmitter.06.BUG | - | 143 | - | 900 | ✓ | 164 | - | 900 | ✓ | 2.79 | - | 900 |
| transmitter.07.BUG | - | 134 | - | 900 | ✓ | 858 | - | 900 | ✓ | 3.65 | - | 900 |
| transmitter.08.BUG | - | 131 | - | 900 | - | 900 | - | 900 | ✓ | 5.23 | - | 900 |
| transmitter.09.BUG | - | 124 | - | 900 | - | 900 | - | 900 | ✓ | 10.3 | - | 900 |
| transmitter.10.BUG | - | 148 | - | 900 | - | 900 | - | 900 | ✓ | 28.0 | - | 900 |
| transmitter.11.BUG | - | 130 | - | 900 | - | 900 | - | 900 | ✓ | 124 | - | 900 |
| transmitter.12.BUG | - | 117 | - | 900 | - | 900 | - | 900 | ✓ | 501 | - | 900 |

Continued on next page

**Table 3.1 – continued from previous page**

| | CBMC $k = \infty$ | | CBMC $k = 10$ | | SATABS | | BLAST | | Explicit | | Predicate LBE | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `transmitter.13.BUG` | - | 116 | - | 900 | - | 900 | - | 900 | - | 900 | - | 900 |
| `transmitter.15.BUG` | - | 900 | - | 900 | ✓ | 226 | - | 900 | ✓ | 529 | ✓ | 2.22 |
| `transmitter.16.BUG` | - | 900 | - | 900 | ✓ | 224 | - | 900 | - | 900 | ✓ | 2.35 |
| `bist_cell` | - | 900 | - | 0.77 | ✓ | 13.0 | ✓ | 279 | ✓ | 1.47 | ✓ | 206 |
| `kundu` | - | 900 | - | 122 | ✓ | 35.4 | - | 84.8 | ✓ | 6.05 | - | 900 |
| `mem_slave_tlm.1` | - | 900 | - | 900 | ✓ | 22.9 | - | 536 | - | 2.17 | - | 900 |
| `mem_slave_tlm.2` | - | 900 | - | 900 | ✓ | 62.7 | - | 378 | - | 2.46 | - | 900 |
| `mem_slave_tlm.3` | - | 900 | - | 900 | ✓ | 147 | - | 236 | - | 2.73 | - | 900 |
| `mem_slave_tlm.4` | - | 900 | - | 900 | ✓ | 282 | - | 219 | - | 3.18 | - | 900 |
| `mem_slave_tlm.5` | - | 900 | - | 900 | ✓ | 568 | - | 245 | - | 3.58 | - | 900 |
| `pc_sfifo_1` | - | 900 | - | 12.5 | ✓ | 1.55 | ✓ | 7.27 | - | 900 | ✓ | 4.88 |
| `pc_sfifo_2` | - | 900 | - | 6.33 | ✓ | 1.92 | ✓ | 41.2 | - | 900 | ✓ | 10.3 |
| `pc_sfifo_3` | - | 900 | - | 7.53 | ✓ | 0.48 | ✓ | 6.22 | ✓ | 1.63 | ✓ | 4.93 |
| `pipeline` | - | 900 | - | 900 | - | 900 | - | 151 | - | 11.2 | - | 900 |
| `token_ring.01` | - | 76.7 | - | 16.2 | ✓ | 4.51 | ✓ | 25.0 | - | 1.61 | ✓ | 7.97 |
| `token_ring.02` | - | 101 | - | 71.6 | ✓ | 14.3 | ✓ | 167 | - | 1.82 | - | 900 |
| `token_ring.03` | - | 104 | - | 231 | ✓ | 33.2 | - | 491 | - | 2.17 | - | 900 |
| `token_ring.04` | - | 128 | - | 623 | ✓ | 95.2 | - | 900 | - | 2.67 | - | 900 |
| `token_ring.05` | - | 107 | - | 900 | ✓ | 208 | - | 900 | - | 3.66 | - | 900 |
| `token_ring.06` | - | 125 | - | 900 | ✓ | 615 | - | 900 | - | 5.53 | - | 900 |
| `token_ring.07` | - | 115 | - | 900 | - | 900 | - | 900 | - | 8.84 | - | 900 |
| `token_ring.08` | - | 118 | - | 900 | - | 900 | - | 900 | - | 27.4 | - | 900 |
| `token_ring.09` | - | 120 | - | 900 | - | 900 | - | 900 | - | 96.3 | - | 900 |
| `token_ring.10` | - | 122 | - | 900 | - | 900 | - | 900 | - | 486 | - | 900 |
| `token_ring.11` | - | 126 | - | 900 | - | 900 | - | 900 | - | 900 | - | 900 |
| `token_ring.12` | - | 119 | - | 900 | - | 900 | - | 900 | - | 900 | - | 900 |
| `token_ring.13` | - | 124 | - | 900 | - | 900 | - | 900 | - | 900 | - | 900 |
| `toy` | - | 900 | - | 30.8 | ✓ | 6.17 | - | 900 | - | 2.34 | - | 900 |
| **SystemC total** | - | 24000 | 5 | 36700 | 39 | 25800 | 15 | 28000 | 20 | 14500 | 18 | 41300 |
| **Total** | 9 | 37500 | 21 | 36900 | 81 | 34100 | 50 | 31800 | 58 | 19900 | 64 | 41900 |

The fifth column lists the results for BLAST 2.5 with options (`-craig 2 -dfs -predH 7 -nosimplemem -alias ""`). (`-craig 2`) specifies to use local tables of predicates with craig interpolants to generate predicates, (`-dfs`) specifies to use depth first search, (`-predH 7`) uses the recommended predicate discovery heuristic, and (`-nosimplemem -alias ""`) disables alias analysis that we do not use for other analyses [31]. We use MATHSAT as solver for BLAST experiments [17]. The sixth column lists results for the *explicit analysis CPA* that we run using the config file (`explicitAnalysis.properties`) from CPACHECKER repository. This configuration uses CBMC to verify the feasibility of error paths. The rightmost column lists results for *predicate analysis CPA with* LBE. We run predicate analysis with (`predicateAbstractionlbe.properties`) from CPACHECKER repository [27].

In the table, after each benchmark set, a sub-total is given for the performance of a tool on that set of benchmark programs. This row shows the number of successful analyses done by the tool and give the total time spent in seconds. The first sub-column in each column shows the analysis result and the second sub-column shows the total time spent on the program. At the bottom row, a summary of results is given in terms of number of successful analyses and total time spent. The time limit for all programs is set to 900 s. Note that some tools usually start an external process and the total time spent by these tools are updated after this external process terminates. In some cases, the tool times out before the process is terminated and the time is not updated properly. We adjusted these results to 900 s in the case of a time-out by a program. So, all the failing results with 900 s indicate that the tool timed out and crashed. None of the tools reported a false positive or a false negative result. There are a total of 108 programs that we analyzed.

**Evaluation.** Comparing CBMC results clearly demonstrates that we should specify a loop bound for CBMC since it is almost certain that there will be (possibly deep) loops or re-entrant edges in the real programs. CBMC with loop bound $k = 10$ can analyze 12 more examples than CBMC with no fixed loop bounds. The configuration with loop bound $k = 10$ for CBMC is very effective in detecting bugs in the benchmark set `ntdrivers`. It also verifies the programs without bugs except `diskperf_simpl1`. It also runs 6 of the `ssh` programs successfully. It can be observed that, when CBMC is successful in analyzing a program, it reports the result in less than 1 second. For `ntdrivers`, CBMC with $k = 10$ spends only 4.4 s and can analyze 9 out of 10 programs.

BLAST performs the predicate analysis using SBE approach. It can analyze 50 programs successfully, performing better than CBMC. When the number of branchings get multiplied

exponentially by adding another condition to the `locks` examples, the ARG gets huge and BLAST takes longer for `locks` benchmark with larger numbered versions due to its SBE approach for predicate analysis. BLAST performs well overall for `ntdrivers` and `ssh` programs. In many cases, predicate analysis with LBE using CPACHECKER outperformed BLAST.

*Explicit analysis* can run 9 more programs than BLAST. Since it explicitly tracks assignments to integer variables by running a path-sensitive analysis, it also suffers from state space explosion when the number of branches in the CFA gets bigger. It needs to build the ARG for each path separately and `locks` experiments demonstrate the behaviour clearly: When the number of branches gets larger, the run-time grows significantly. One observation on *explicit analysis* is that, when it can analyze a program successfully, it usually takes a few seconds to terminate. It also reports UNKNOWN after a few seconds for many programs since it is not precise enough to verify the program or find a feasible error path. For example, for `ssh` programs, it spends significantly less time than other tools and can report conclusive results for 19 programs.

*Predicate analysis* with LBE is the best configuration overall for `locks`, `ntdrivers` and `ssh` examples. It can run all of the programs in these benchmark sets successfully and spends significantly less time compared to other tools. SATABS is also very fast with `locks` but it is slower for `ntdrivers` and `ssh` programs and times out for some of them. SATABS is very powerful on the benchmark set `systemc`. It can analyze 39 of them successfully whereas BLAST achieves to analyze 15, CPACHECKER with *explicit analysis* 20 and CPACHECKER with *predicate analysis* 18. Due to large number of `systemc` benchmark programs, SATABS can run more programs successfully in total, although it should be noted that CPACHECKER with *explicit analysis* and with *predicate analysis* can successfully analyze a different set of programs from `systemc` benchmarks. *Explicit analysis* is especially effective for `transmitter`* programs. We will use this result to improve results of CPACHECKER in upcoming chapters.

Even though we have included total run-times for tools, it is debatable how useful the total run-time values for evaluating tools. A tool might crash for a program after a few seconds and the other tool might analyze the program for the entire time limit allocated and terminate after 900 s. It might actually be desirable for the developer to run the analysis for the entire time given to the tool instead of crashing with an unhandled case in the tool. Thus, the number of analyzed programs might be a more useful comparison parameter.

Investigating the reason of SATABS's better performance for the benchmark set `systemc`, we have seen that using different approaches for abstraction and refinement was the main reason of the difference. SATABS used an approach that inlined functions while running the predicate analysis whereas we used ABE approach that computes the abstraction on function entry nodes and loop heads. We have an extension of the ABE approach with function inlining that is implemented by Wendler in his Master's thesis, *Software verification based on adjustable large-block encoding* [106]. It is not in the scope of this thesis so we will not explain the details but provide a summary on how it works as follows: This approach inlines functions and updates the path formula by appending the path formula of a function to the current path formula instead of computing the abstraction at the entry of the function. When we used ABE with function inlining, we had similar results to SATABS for some `systemc` programs, such as `pipeline`. This program cannot be analyzed by pure ABE approach in 15 min time limit, but with function inlining, it is analyzed in less than 15 s. The function inlining approach with ABE might have memory consumption problems causing tool crashes [106]. So, we use ABE as our standard predicate analysis approach throughout the thesis.

# Chapter 4

# Sequential Composition of CPAs

In this chapter, we present a framework that uses several CPAs not in parallel as in the CPA framework explained in Chapter 2, but separately in an order based on each CPA's strengths. As we have explained in previous chapters, all analyses have their characteristic properties and they are better than other analyses at extracting information about programs with some specific properties. For example, some tools are good at detecting shallow bugs without exploring the state space of the program in depth. On the other hand, these techniques might not be powerful enough to show the safety of the program by exploring all abstract states of the program within reasonable time and memory requirements. We introduced and implemented a framework that limits an analysis with some conditions and based on the results of the analysis, continues with another configuration.

## 4.1 Behaviour of Different Analysis Techniques

We have explained some analysis techniques and their properties in Chapter 2. CBMC is a verification tool that runs a bit-precise bounded model checking algorithm [44, 59]. As demonstrated in Table 3.1, when it detects any bugs, it can identify them after spending very little time and it did not report a result for many programs that do not have a bug in the given time limits. *Explicit analysis CPA* is a path-sensitive data-flow analysis. On programs that have many assignments to integers, explicit analysis is more efficient. It is also important that the assigned values are known in compile time to ensure its efficiency. For instance, if there are two variables that have been assigned to random values and compared in an assume operation, *explicit analysis CPA* will not be able to capture the relation imposed

by the assume operation. It only keeps a mapping from variables to values and if this map does not grow very fast, explicit analysis will be useful especially for identifying bugs faster than predicate analysis [27]. *Predicate abstraction* using LBE is very efficient and compared to traditional predicate abstraction with SBE [17, 28]. It has also a more expressive power compared to the *explicit analysis CPA* since any relationship between variables can be modeled using appropriate predicates. Also, on-the-fly refinement and abstraction makes it a good candidate for building the abstract state space of the programs that do not contain bugs.

### 4.1.1 Imprecision of Explicit Analysis

We present an example for a C program that is analyzed using explicit analysis and the feasibility of the error path is checked by a more precise algorithm (such as CBMC algorithm as explained in Section 3.3.1). Explicit analysis may not be precise enough to model relations between variables if there is no explicit assignment to this variable in the program. That may result in many false alarms due to the explicit analysis' imprecision. When an infeasible error path is reported by the explicit analysis, we use CBMC to check the feasibility of the error path. If the error path is found to be infeasible (i. e., the error is not actually reachable following the reported path), the analysis continues with other elements in waitlist and ignores this path. We hope, eventually the analysis will report a real error which is also validated by the precise path analysis. If no real errors are detected at the end of the analysis, ignoring the error paths which were found to be infeasible makes the analysis imprecise (unsoundness). This is because the ignored path may have some states that cover other states on the path to the error and they are not explored before termination.

This case is illustrated by a simple program shown in Figure 4.1.

At lines 3 and 4, a random value is assigned to integer variables `x` and `y`. Conditions given at lines 6 and 9, assigns a new value to the variable `y` based on the value of `x` and conditions given at 13 and 16 checks whether the assignments satisfy the expected conditions. Note that the case for (`x` = 5) is ignored when checking the values of `y`, so the assertion at line 17 should fail. The CFA for the program is given in Figure 4.2. When we run explicit analysis with default configuration file on the program, the ARG given in Figure 4.3 is produced. The abstract states are modeled as nodes and shown in the dashed boxes on the ARG. As shown in Figure 4.3, first the if branch is taken during the analysis and the abstract node 4 is computed before node 5. The analysis will continue with the abstract node shown with

```
1  int main(){
2
3     int x = nondet_int();
4     int y = nondet_int();
5
6     if(x > 5){
7        y = x * 3;
8     }
9     else{
10       y = x * 2;
11    }
12
13    if(x > 5){
14      assert(y > 15);
15    }
16    else{
17      assert(y < 10);
18    }
19
20    return (0);
21 }
```

Figure 4.1: Example program producing an imprecise result with the explicit analysis

id 4 and the node 5 will remain in waitlist. Since there are no assignments to variables, the explicit analysis does not save any values for `x` and `y` and the abstract states for all nodes are $\top$. The analysis continues with the node 6a, and it reaches to the error label specified by the first assertion (`assert(y > 15);`) from the if branch. The error state is shown with node 8 on the ARG. This path is reachable for the explicit analysis because the abstract state is $\top$ for the error state.

In fact, this path is not reachable in the concrete state space of the program because when `x` $> 5$, then `y` is guaranteed to be larger than 15. This false alarm can be refuted by a more precise path analysis and the process will continue with the states in the set waitlist. The analysis continues from abstract state shown with node 5. The next abstract state computed for location 6 will be 6b. Since there are no assignments in this branch too, 6b is also $\top$. This means that the newly computed state is covered by 6a and the analysis stops. Since there are no more elements in waitlist, the analysis will terminate before checking the reachability of the second assertion, (`assert(y < 10);`). So, even though it was possible to reach the error label through the else branch, since the path was falsified by a precise path feasibility check algorithm, the analysis was not able to reach the error label.

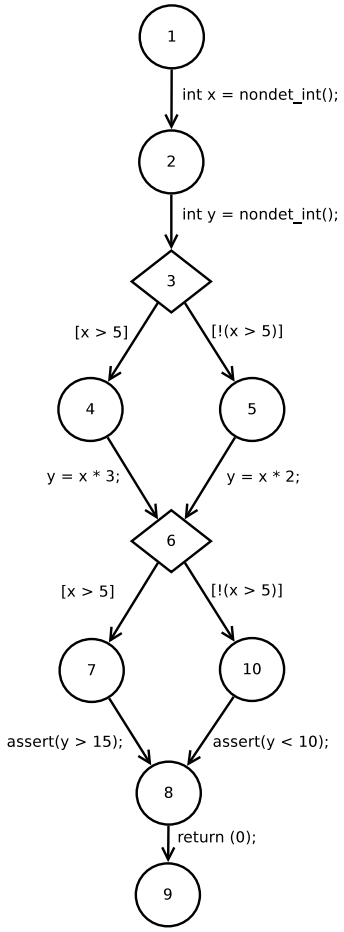


Figure 4.2: CFA for the example in Figure 4.1

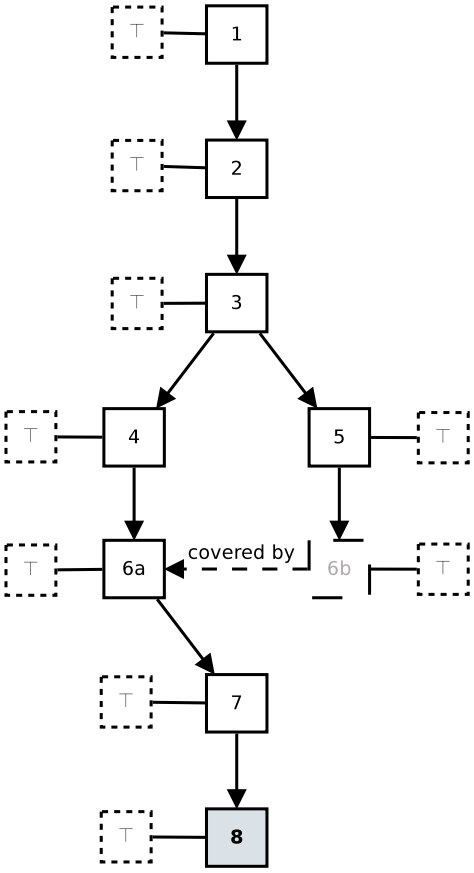


Figure 4.3: ARG for the example in Figure 4.1

This illustrates that even though we have a real bug in the program, it might not be reached if the CBMC validation detects a false alarm during the analysis. If such a case is encountered, the analysis result is imprecise. The imprecision of the explicit analysis due to its lack of support for representing relations between variables results in many UNKNOWN answers reported by it.

## 4.2 Implementation of a Sequential Composition Framework

Observing different behaviour of software analysis techniques, we extended CPAchecker to provide mechanisms to easily integrate several analysis together to run in a sequential

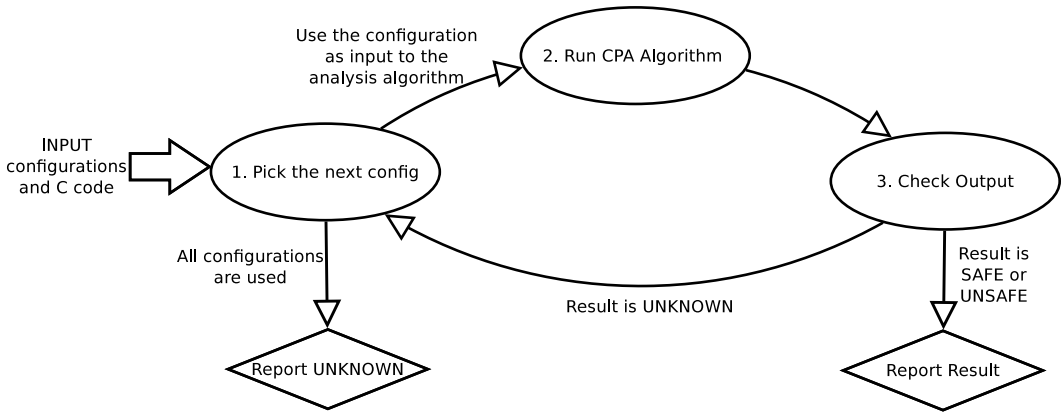


Figure 4.4: Design Overview of Sequential Composition

order. Since the CPA framework and the analysis algorithm defines a software analysis, we did not want to alter definitions and interfaces for them. Instead, we created another algorithm that wraps the CPA algorithm and uses it as a separate component. That lets us use the CPA algorithm separately without introducing any changes to it.

Figure 4.4 shows the overview of the design for sequential composition algorithm. The algorithm takes the source code of the program as input and a list of configurations. These configurations define which algorithm will be used next and define each algorithm's parameters and options. For example, if the user wants to use the *explicit analysis CPA* first with a time limit and then switch to *predicate analysis CPA* after the first analysis returns UNKNOWN, then the list of configurations contains the *explicit analysis CPA* with the specified time limit and the *predicate analysis CPA*. The algorithm runs the analysis for the selected configuration as the next step. When the analysis terminates, the output is inspected and if the result is SAFE or UNSAFE, then the result is reported to the user. Since we require that all analyses produce sound approximations of the concrete state space, a SAFE answer will conclude that the analysis did not find the specified error condition in the program. An UNSAFE answer will report the error path and terminate the analysis. If the analysis finds out that all configurations are used from the input configurations list before reporting a result, the analysis terminates and reports UNKNOWN.

**Algorithm 3** *SequentialComposition*($L$)

**Input:** a list $L$ containing a list of algorithm configurations to use,
**Output:** a set of reachable abstract states,
**Variables:** two sets reached and waitlist

```
1:  while true do
2:     reached := create a new set;
3:     waitlist := create a new set;
4:     pop next algorithm configuration A from the list L
5:     (reached, waitlist) := A.run(reached, waitlist);
6:     //report ERROR
7:     if reached contains an ERROR state then
8:        break;
9:     //report SAFE
10:    if waitlist has no elements and reached is sound then
11:       break;
12:    // report UNKNOWN
13:    if L == ∅ then
14:       break;
15: return (reached, waitlist)
```

We show the implementation of the sequential composition algorithm in Algorithm 3. The algorithm is very intuitive; it starts with a list of configurations and at each iteration of the main loop, the next configuration is selected and run. Each configuration in the list $L$ provides an algorithm that extends the CPA algorithm interface explained in Chapter 2. Note that we also use external tools such as CBMC to run an algorithm and since we use them as external tools, we have to run them such that they produce a result which is compatible with the CPA algorithm. Such an external call is wrapped by a method and it gets the same input as a CPA algorithm. An algorithm that runs CBMC inside the CPAchecker tool was explained in Chapter 3.

Algorithm 3 may terminate in three cases after the selected configuration is run:

1. If an ERROR element is found in the set reached, the analysis terminates and reports the detected safety violation (line 7).

2. If there are no elements left in the set waitlist to process and reached reports that the analysis was SAFE, then all possible abstract states are discovered without finding a bug and the analysis reports SAFE (line 10).

3. An analysis was finished without reporting UNSAFE but the analysis was not sound, and there are no other configurations in the list $L$ to proceed with. Then, the analysis reports UNKNOWN and outputs the sets reached and waitlist (line 13).

Later, we will demonstrate how the unsoundness can be useful to improve the verification results when we run the sequential composition algorithm. In this chapter, we simply introduce the benefits of using different analysis techniques in a sequential order. Lines 2 and 3 of Algorithm 3 shows that before each call to a different analysis algorithm, we create a new reached set and waitlist for the analysis. Later, we will also show that this step can be improved by not creating a new set but by using the outputs of the previous algorithm to proceed with.

## 4.3 Experiments Using Sequential Composition

We created several sequential composition configurations and run several experiments to test the performance of CPAchecker by comparing different analyses. The configurations that we created and tested are configured by each used analysis' properties. Some of them are limited with a small time-limit to achieve the best results and some are given more time. Table 4.1 shows results for analysis with several configurations. The structure of the table is similar to the structure of Table 3.1. We used 15 min time limit unless otherwise stated, and we show whether a tool was successful on the first sub-column and total time used is shown in the second sub-column.

Table 4.1: Using sequential composition

| | CBMC | | | | Explicit | | Explicit | | Predicate | | Comb. A | | Comb. B | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $k = 1$ | | $k = 10$ | | | | $time(10s)$ | | LBE | | Exp + LBE | | CBMC + Expt + LBE | |
| test_locks_14.BUG | ✓ | 0.03 | ✓ | 0.37 | ✓ | 289 | - | 11.1 | ✓ | 1.72 | ✓ | 11.6 | ✓ | 1.40 |
| test_locks_15.BUG | ✓ | 0.03 | - | 0.42 | - | 543 | - | 11.1 | ✓ | 1.77 | ✓ | 11.2 | ✓ | 1.19 |
| test_locks_5 | - | 0.02 | - | 0.07 | ✓ | 1.52 | ✓ | 1.73 | ✓ | 1.62 | ✓ | 1.96 | ✓ | 1.79 |
| test_locks_6 | - | 0.02 | - | 0.09 | ✓ | 2.08 | ✓ | 2.10 | ✓ | 1.35 | ✓ | 2.37 | ✓ | 2.03 |
| test_locks_7 | - | 0.02 | - | 0.12 | ✓ | 1.92 | ✓ | 2.38 | ✓ | 1.61 | ✓ | 2.21 | ✓ | 2.28 |
| test_locks_8 | - | 0.02 | - | 0.14 | ✓ | 2.82 | ✓ | 3.14 | ✓ | 1.66 | ✓ | 3.34 | ✓ | 2.91 |
| test_locks_9 | - | 0.02 | - | 0.16 | ✓ | 3.35 | ✓ | 3.96 | ✓ | 1.41 | ✓ | 4.36 | ✓ | 4.12 |
| test_locks_10 | - | 0.02 | - | 0.21 | ✓ | 6.28 | ✓ | 8.32 | ✓ | 1.41 | ✓ | 7.73 | ✓ | 8.57 |
| test_locks_11 | - | 0.02 | - | 0.24 | ✓ | 25.0 | - | 10.4 | ✓ | 1.42 | ✓ | 10.5 | ✓ | 10.5 |
| test_locks_12 | - | 0.02 | - | 0.27 | ✓ | 122 | - | 10.4 | ✓ | 1.43 | ✓ | 10.5 | ✓ | 10.6 |
| test_locks_13 | - | 0.03 | - | 0.31 | ✓ | 556 | - | 10.4 | ✓ | 1.44 | ✓ | 10.5 | ✓ | 10.5 |
| test_locks_14 | - | 0.03 | - | 0.35 | - | 900 | - | 10.4 | ✓ | 1.65 | ✓ | 10.6 | ✓ | 10.5 |
| test_locks_15 | - | 0.03 | - | 0.40 | - | 900 | - | 10.5 | ✓ | 1.45 | ✓ | 10.5 | ✓ | 10.6 |
| **Locks total** | 2 | 0.31 | 1 | 3.15 | 10 | 3350 | 6 | 95.3 | 13 | 19.9 | 13 | 97.4 | 13 | 77.0 |
| cdaudio_simpl1_BUG | ✓ | 0.99 | ✓ | 1.09 | ✓ | 2.99 | ✓ | 3.24 | ✓ | 9.49 | ✓ | 2.87 | ✓ | 2.67 |
| floppy_simpl3_BUG | ✓ | 0.16 | ✓ | 0.25 | ✓ | 2.47 | ✓ | 2.43 | ✓ | 6.81 | ✓ | 2.31 | ✓ | 1.52 |
| floppy_simpl4_BUG | ✓ | 0.30 | ✓ | 0.40 | ✓ | 2.58 | ✓ | 2.84 | ✓ | 10.2 | ✓ | 2.69 | ✓ | 1.81 |
| kbfiltr_simpl2_BUG | ✓ | 0.11 | ✓ | 0.21 | ✓ | 2.15 | ✓ | 2.40 | ✓ | 4.00 | ✓ | 2.23 | ✓ | 1.42 |
| cdaudio_simpl1 | ✓ | 1.01 | ✓ | 1.13 | ✓ | 3.67 | ✓ | 3.66 | ✓ | 17.2 | ✓ | 3.67 | ✓ | 2.68 |
| diskperf_simpl1 | - | 0.26 | - | 0.35 | - | 900 | - | 10.4 | ✓ | 15.1 | ✓ | 23.3 | ✓ | 23.5 |
| floppy_simpl3 | ✓ | 0.16 | ✓ | 0.25 | ✓ | 2.62 | ✓ | 3.37 | ✓ | 8.66 | ✓ | 3.34 | ✓ | 1.49 |
| floppy_simpl4 | ✓ | 0.28 | ✓ | 0.38 | ✓ | 3.32 | ✓ | 3.81 | ✓ | 12.5 | ✓ | 3.51 | ✓ | 1.98 |
| kbfiltr_simpl1 | ✓ | 0.05 | ✓ | 0.14 | ✓ | 2.39 | ✓ | 2.83 | ✓ | 3.09 | ✓ | 3.16 | ✓ | 1.27 |
| kbfiltr_simpl2 | ✓ | 0.11 | ✓ | 0.20 | ✓ | 2.89 | ✓ | 3.53 | ✓ | 4.57 | ✓ | 3.42 | ✓ | 1.64 |
| **NT drivers total** | 9 | 3.43 | 9 | 4.40 | 9 | 925 | 9 | 38.5 | 10 | 91.6 | 10 | 50.5 | 10 | 40.0 |
| s3_clnt_1_BUG | - | 0.03 | - | 4.25 | ✓ | 4.17 | ✓ | 4.79 | ✓ | 3.11 | ✓ | 5.01 | ✓ | 4.48 |
| s3_clnt_2_BUG | - | 0.03 | - | 4.38 | ✓ | 4.15 | ✓ | 4.89 | ✓ | 2.82 | ✓ | 4.49 | ✓ | 4.48 |
| s3_clnt_3_BUG | - | 0.03 | ✓ | 5.34 | ✓ | 3.99 | ✓ | 4.67 | ✓ | 3.21 | ✓ | 4.38 | ✓ | 4.47 |
| s3_clnt_4_BUG | - | 0.03 | - | 4.38 | ✓ | 3.77 | ✓ | 4.50 | ✓ | 3.65 | ✓ | 4.36 | ✓ | 4.50 |

**Table 4.1 – continued from previous page**

| | CBMC | | | | Explicit | | Explicit | | Predicate | | Comb. A | | Comb. B | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $k = 1$ | | $k = 10$ | | | | $time(10s)$ | | LBE | | Exp + LBE | | CBMC + Expt + LBE | |
| s3_srvr_11_BUG | - | 0.03 | ✓ | 6.73 | ✓ | 2.57 | ✓ | 2.46 | ✓ | 3.60 | ✓ | 2.32 | ✓ | 2.46 |
| s3_srvr_12_BUG | - | 0.04 | ✓ | 8.62 | ✓ | 2.24 | ✓ | 2.38 | ✓ | 5.62 | ✓ | 2.35 | ✓ | 2.83 |
| s3_srvr_14_BUG | - | 0.03 | - | 6.35 | ✓ | 2.60 | ✓ | 2.90 | ✓ | 2.48 | ✓ | 2.76 | ✓ | 3.37 |
| s3_srvr_1_BUG | - | 0.03 | ✓ | 6.55 | ✓ | 1.83 | ✓ | 1.92 | ✓ | 2.76 | ✓ | 2.08 | ✓ | 1.91 |
| s3_srvr_2_BUG | - | 0.03 | ✓ | 6.29 | ✓ | 2.09 | ✓ | 2.17 | ✓ | 2.85 | ✓ | 2.09 | ✓ | 2.18 |
| s3_srvr_6_BUG | ✓ | 0.03 | ✓ | 7.08 | ✓ | 274 | - | 10.4 | ✓ | 1.63 | ✓ | 10.5 | ✓ | 1.74 |
| s3_clnt_1 | - | 0.03 | - | 4.36 | ✓ | 8.07 | ✓ | 10.1 | ✓ | 7.62 | ✓ | 17.3 | ✓ | 16.6 |
| s3_clnt_2 | - | 0.03 | - | 4.65 | ✓ | 7.86 | ✓ | 9.63 | ✓ | 6.72 | ✓ | 16.3 | ✓ | 15.8 |
| s3_clnt_3 | - | 0.03 | - | 5.38 | ✓ | 8.99 | ✓ | 10.4 | ✓ | 5.82 | ✓ | 14.3 | ✓ | 14.4 |
| s3_clnt_4 | - | 0.03 | - | 4.81 | ✓ | 8.29 | ✓ | 9.76 | ✓ | 10.4 | ✓ | 18.2 | ✓ | 18.6 |
| s3_srvr_1 | - | 0.03 | - | 4.15 | ✓ | 2.58 | ✓ | 3.09 | ✓ | 21.6 | ✓ | 2.95 | ✓ | 3.02 |
| s3_srvr_1a | - | 0.02 | - | 0.23 | - | 2.91 | - | 2.75 | ✓ | 2.82 | ✓ | 4.54 | ✓ | 4.53 |
| s3_srvr_1b | - | 0.01 | - | 0.15 | - | 1.84 | - | 1.89 | ✓ | 1.71 | ✓ | 2.03 | ✓ | 2.37 |
| s3_srvr_2 | - | 0.03 | - | 5.58 | ✓ | 2.87 | ✓ | 3.04 | ✓ | 149 | ✓ | 3.46 | ✓ | 3.03 |
| s3_srvr_3 | - | 0.03 | - | 5.68 | - | 2.73 | - | 3.29 | ✓ | 9.49 | ✓ | 11.9 | ✓ | 11.0 |
| s3_srvr_4 | - | 0.03 | - | 5.64 | - | 2.70 | - | 3.53 | ✓ | 28.6 | ✓ | 28.3 | ✓ | 29.3 |
| s3_srvr_6 | - | 0.03 | - | 6.64 | ✓ | 241 | - | 10.4 | ✓ | 228 | ✓ | 233 | ✓ | 232 |
| s3_srvr_7 | - | 0.03 | - | 6.37 | ✓ | 180 | - | 10.4 | ✓ | 46.2 | ✓ | 54.1 | ✓ | 55.3 |
| s3_srvr_8 | - | 0.03 | - | 6.13 | ✓ | 2.52 | ✓ | 3.13 | ✓ | 21.7 | ✓ | 3.00 | ✓ | 3.14 |
| **SSH total** | 1 | 0.67 | 6 | 120 | 19 | 774 | 16 | 122 | 23 | 571 | 23 | 450 | 23 | 442 |
| kundu1_BUG | - | 0.02 | - | 18.0 | ✓ | 2.24 | ✓ | 2.18 | ✓ | 15.1 | ✓ | 2.06 | ✓ | 2.18 |
| kundu2_BUG | - | 0.03 | - | 127 | ✓ | 1.89 | ✓ | 1.71 | ✓ | 829 | ✓ | 1.64 | ✓ | 1.74 |
| pc_sfifo_1_BUG | - | 0.02 | ✓ | 13.1 | - | 900 | - | 10.6 | ✓ | 1.43 | ✓ | 10.7 | ✓ | 10.8 |
| pc_sfifo_2_BUG | - | 0.03 | ✓ | 7.11 | - | 900 | - | 10.7 | ✓ | 1.46 | ✓ | 11.0 | ✓ | 10.8 |
| pipeline_BUG | - | 0.05 | - | 900 | - | 11.0 | - | 10.6 | - | 900 | - | 900 | - | 900 |
| token_ring.01.BUG | - | 0.03 | - | 22.0 | - | 1.95 | - | 1.89 | ✓ | 3.72 | ✓ | 3.91 | ✓ | 4.46 |
| token_ring.02.BUG | - | 0.03 | - | 82.0 | - | 2.06 | - | 2.21 | ✓ | 32.9 | ✓ | 34.5 | ✓ | 34.4 |
| token_ring.03.BUG | - | 0.04 | - | 263 | - | 2.60 | - | 2.69 | - | 900 | - | 900 | - | 900 |
| token_ring.04.BUG | - | 0.03 | - | 653 | - | 2.90 | - | 3.43 | - | 900 | - | 900 | - | 900 |
| token_ring.05.BUG | - | 0.05 | - | 900 | - | 3.81 | - | 4.45 | - | 900 | - | 900 | - | 900 |
| token_ring.06.BUG | - | 0.05 | - | 900 | - | 4.94 | - | 6.05 | - | 900 | - | 900 | - | 900 |

**Table 4.1 – continued from previous page**

| | CBMC | | | | Explicit | | Explicit | | Predicate | | Comb. A | | Comb. B | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $k = 1$ | | $k = 10$ | | | | $time(10s)$ | | LBE | | Exp + LBE | | CBMC + Expt + LBE | |
| token_ring.07.BUG | - | 0.05 | - | 900 | - | 8.83 | - | 10.6 | - | 900 | - | 900 | - | 900 |
| token_ring.08.BUG | - | 0.05 | - | 900 | - | 29.1 | - | 10.8 | - | 900 | - | 900 | - | 900 |
| token_ring.09.BUG | - | 0.06 | - | 900 | - | 125 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| token_ring.10.BUG | - | 0.06 | - | 900 | - | 501 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| token_ring.11.BUG | - | 0.06 | - | 900 | - | 900 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| token_ring.12.BUG | - | 0.06 | - | 900 | - | 900 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| token_ring.13.BUG | - | 0.08 | - | 900 | - | 900 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| token_ring.14.BUG | - | 0.07 | - | 900 | ✓ | 498 | - | 10.6 | - | 900 | - | 900 | - | 900 |
| token_ring.15.BUG | - | 0.07 | - | 900 | - | 900 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| toy1_BUG | - | 0.04 | - | 34.7 | - | 2.78 | - | 3.02 | ✓ | 345 | ✓ | 346 | ✓ | 354 |
| toy2_BUG | - | 0.04 | - | 31.2 | ✓ | 2.21 | ✓ | 2.30 | ✓ | 122 | ✓ | 2.59 | ✓ | 2.37 |
| transmitter.01.BUG | - | 0.02 | ✓ | 5.29 | ✓ | 1.55 | ✓ | 1.94 | ✓ | 2.03 | ✓ | 1.59 | ✓ | 1.98 |
| transmitter.02.BUG | - | 0.03 | ✓ | 22.4 | ✓ | 1.73 | ✓ | 2.11 | ✓ | 4.89 | ✓ | 1.75 | ✓ | 1.90 |
| transmitter.03.BUG | - | 0.03 | - | 94.1 | ✓ | 1.97 | ✓ | 2.31 | ✓ | 34.5 | ✓ | 2.32 | ✓ | 2.08 |
| transmitter.04.BUG | - | 0.03 | - | 269 | ✓ | 2.26 | ✓ | 2.70 | - | 900 | ✓ | 2.30 | ✓ | 2.31 |
| transmitter.05.BUG | - | 0.04 | ✓ | 673 | ✓ | 2.46 | ✓ | 2.51 | - | 900 | ✓ | 2.61 | ✓ | 2.57 |
| transmitter.06.BUG | - | 0.04 | - | 900 | ✓ | 2.79 | ✓ | 3.05 | - | 900 | ✓ | 3.05 | ✓ | 3.12 |
| transmitter.07.BUG | - | 0.05 | - | 900 | ✓ | 3.65 | ✓ | 3.75 | - | 900 | ✓ | 3.90 | ✓ | 4.48 |
| transmitter.08.BUG | - | 0.05 | - | 900 | ✓ | 5.23 | ✓ | 5.81 | - | 900 | ✓ | 6.14 | ✓ | 6.19 |
| transmitter.09.BUG | - | 0.06 | - | 900 | ✓ | 10.3 | ✓ | 10.5 | - | 900 | ✓ | 10.5 | ✓ | 10.8 |
| transmitter.10.BUG | - | 0.07 | - | 900 | ✓ | 28.0 | - | 11.7 | - | 900 | - | 900 | - | 900 |
| transmitter.11.BUG | - | 0.07 | - | 900 | ✓ | 124 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| transmitter.12.BUG | - | 0.07 | - | 900 | ✓ | 501 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| transmitter.13.BUG | - | 0.07 | - | 900 | - | 900 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| transmitter.15.BUG | - | 0.08 | - | 900 | ✓ | 529 | - | 10.5 | ✓ | 2.22 | ✓ | 11.6 | ✓ | 11.7 |
| transmitter.16.BUG | - | 0.08 | - | 900 | - | 900 | - | 11.0 | ✓ | 2.35 | ✓ | 11.9 | ✓ | 11.8 |
| bist_cell | - | 0.03 | - | 0.77 | ✓ | 1.47 | ✓ | 1.45 | ✓ | 206 | ✓ | 1.49 | ✓ | 1.55 |
| kundu | - | 0.03 | - | 122 | ✓ | 6.05 | ✓ | 7.94 | - | 900 | ✓ | 8.02 | ✓ | 7.50 |
| mem_slave_tlm.1 | - | 0.04 | - | 900 | - | 2.17 | - | 2.14 | - | 900 | - | 900 | - | 900 |
| mem_slave_tlm.2 | - | 0.05 | - | 900 | - | 2.46 | - | 2.87 | - | 900 | - | 900 | - | 900 |
| mem_slave_tlm.3 | - | 0.05 | - | 900 | - | 2.73 | - | 2.83 | - | 900 | - | 900 | - | 900 |

**Table 4.1 – continued from previous page**

| | CBMC | | | | Explicit | | Explicit | | Predicate | | Comb. A | | Comb. B | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $k = 1$ | | $k = 10$ | | | | $time(10s)$ | | LBE | | Exp + LBE | | CBMC + Expt + LBE | |
| mem_slave_tlm.4 | - | 0.06 | - | 900 | - | 3.18 | - | 3.26 | - | 900 | - | 900 | - | 900 |
| mem_slave_tlm.5 | - | 0.06 | - | 900 | - | 3.58 | - | 3.86 | - | 900 | - | 900 | - | 900 |
| pc_sfifo_1 | - | 0.02 | - | 12.5 | - | 900 | - | 10.7 | ✓ | 4.88 | ✓ | 15.3 | ✓ | 14.1 |
| pc_sfifo_2 | - | 0.02 | - | 6.33 | - | 900 | - | 10.7 | ✓ | 10.3 | ✓ | 18.3 | ✓ | 19.1 |
| pc_sfifo_3 | - | 0.03 | - | 7.53 | ✓ | 1.63 | ✓ | 1.67 | ✓ | 4.93 | ✓ | 1.97 | ✓ | 1.74 |
| pipeline | - | 0.04 | - | 900 | - | 11.2 | - | 10.7 | - | 900 | - | 900 | - | 900 |
| token_ring.01 | - | 0.03 | - | 16.2 | - | 1.61 | - | 1.99 | ✓ | 7.97 | ✓ | 8.69 | ✓ | 9.15 |
| token_ring.02 | - | 0.03 | - | 71.6 | - | 1.82 | - | 1.95 | - | 900 | - | 900 | - | 900 |
| token_ring.03 | - | 0.04 | - | 231 | - | 2.17 | - | 2.37 | - | 900 | - | 900 | - | 900 |
| token_ring.04 | - | 0.04 | - | 623 | - | 2.67 | - | 2.88 | - | 900 | - | 900 | - | 900 |
| token_ring.05 | - | 0.05 | - | 900 | - | 3.66 | - | 4.66 | - | 900 | - | 900 | - | 900 |
| token_ring.06 | - | 0.05 | - | 900 | - | 5.53 | - | 5.57 | - | 900 | - | 900 | - | 900 |
| token_ring.07 | - | 0.06 | - | 900 | - | 8.84 | - | 9.45 | - | 900 | - | 900 | - | 900 |
| token_ring.08 | - | 0.05 | - | 900 | - | 27.4 | - | 10.6 | - | 900 | - | 900 | - | 900 |
| token_ring.09 | - | 0.06 | - | 900 | - | 96.3 | - | 10.7 | - | 900 | - | 900 | - | 900 |
| token_ring.10 | - | 0.06 | - | 900 | - | 486 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| token_ring.11 | - | 0.07 | - | 900 | - | 900 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| token_ring.12 | - | 0.07 | - | 900 | - | 900 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| token_ring.13 | - | 0.07 | - | 900 | - | 900 | - | 10.5 | - | 900 | - | 900 | - | 900 |
| toy | - | 0.03 | - | 30.8 | - | 2.34 | - | 2.41 | - | 900 | - | 900 | - | 900 |
| **SystemC total** | - | 2.95 | 5 | 36700 | 20 | 14500 | 15 | 409 | 18 | 41300 | 25 | 33800 | 25 | 33800 |
| **Total** | 12 | 7.36 | 21 | 36900 | 58 | 19900 | 46 | 666 | 64 | 41900 | 71 | 34400 | 71 | 34400 |

There are results for four additional configurations in this table. We added results for CBMC with loop bound $k = 1$ to compare them to the results with $k = 10$ from Table 3.1. Results for CBMC with loop bound $k = 10$ is shown in the second column. The third column shows explicit analysis results with 15 min time limit from Table 3.1. The forth column shows explicit analysis results with 10 s time limit. They report different performance results because we use an additional CPA to monitor the progress of the explicit analysis with 10 s time limit. We will explain how the monitoring is implemented in Chapter 5. We can use explicit analysis in a sequential combination by the help of the monitoring CPA. Monitoring introduces a small overhead as seen in the table, so it can be ignored for now. Some reported run-times are slightly larger than 10 s because the 10 s limit is only for the analysis and does not include pre and post processing times. The biggest time spent on an analysis is 11.7 s for the explicit analysis with the 10 s limit. The fifth column reports results for predicate analysis using LBE approach from Table 3.1. Last two columns show results for sequential combinations of analyses. In the first combination, denoted as Combination A; first, explicit analysis with 10 s limit and then predicate analysis with LBE is used. The Combination B sequentially composes CBMC with $k = 1$, explicit analysis with 10 s limit and, predicate analysis with LBE.

Comparing performance results for CBMC with the loop bounds $k = 1$ and $k = 10$ reveals that using the loop bound $k = 1$ is very fast. Running all 108 programs takes only 7.36 seconds and it can successfully check 12 programs. Increasing the bound to 10 takes significantly more time, 36900 seconds and can verify only 9 more programs than the configuration with loop bound 1. CBMC with $k = 1$ takes less than 0.1 second except for `diskperf_simpl1` which took 0.26 s to run, when it fails to find an error. For detecting errors or verifying the program, it is extremely fast also. It takes 3.43 s to check 9 out of 10 `ntdrivers` benchmark successfully.

The same comparison can be made for explicit analysis results. The explicit analysis with no conditions shown on the third column can run 58 programs successfully and a majority of them take less than 10 s. So, in fact, if the explicit analysis can analyze a program within the given limits, it is very probable that the analysis will be terminated in a few seconds. Otherwise, since it runs a path-sensitive analysis without using a real abstraction, it is likely to run into an infinite loop for the explicit analysis. When the time is limited to 10 seconds, explicit analysis still verifies or finds a bug for 46 programs. The major improvement is in

total run-time. Using a 10 s time limit takes only 666 seconds whereas the 15 min time limit took 19900 s.

Predicate analysis with LBE is slower but more precise compared to the explicit analysis. Its total run-time is 41900 s but can verify or find bugs for 64 programs. It usually takes a longer time to verify a program due to its 'abstraction, model checking and refinement' loop. But due to its power to represent more relations than the explicit analysis, it is more precise and can analyze more programs successfully. An interesting point is that the explicit analysis and predicate analysis can analyze a different set of programs. For example, `diskperf_simpl1` cannot be verified by the explicit analysis but predicate analysis verifies it in 15.1 s. On the other hand, the error label in `transmitter.04.BUG` can be detected by the explicit analysis in 2.70 s but predicate analysis fails to terminate for this program in the given time limit.

So, the power of different analysis and their characteristic properties can be used to create a configuration for sequentially combining them. Combination A, shown on the column *Comb. A* combines explicit analysis with 10 s time limit and the predicate analysis with LBE. The benefit is from their characteristic properties: As explained above, the explicit analysis usually terminates with a meaningful result in a few seconds most of the time. On the other hand, predicate analysis usually requires more time but it is more powerful and can analyze more programs. Therefore, we first run the explicit analysis by limiting the time allocated for it and if it returns an answer, we terminate and report the result. If it fails to return a sound answer in the given time limit, we switch to predicate analysis and run it on the same program. CPAchecker's strength comes from its extendibility and compatibility with different configurations. This configuration can be created using only a config file with no human interaction.

The combination shows an improvement both in terms of total number of analyzed programs and total time spent on running the entire set. The total row at the bottom of Table 4.1 shows that *Comb. A* can analyze a total of 71 programs which is 7 more than predicate analysis itself and 13 more than explicit analysis itself. Improvement on total time spent is more significant: It spends 7500 seconds less than predicate analysis even though it analyzes more programs successfully. For benchmark sets `locks`, `ntdrivers`, and `ssh`; predicate analysis was able to analyze all programs already. *Comb. A* spends more time for `locks` due to LBE's specialized power on these programs but since explicit analysis can analyze some programs faster than LBE for `ntdrivers` and `ssh`, the combination is more

powerful for these benchmark sets in terms of time spent. The real benefit can be observed for the benchmark set `systemc`. Some of these programs can be analyzed only by the explicit analysis or predicate analysis and *Comb. A* brings these together. There are also programs such as `transmitter.03.BUG` that can be analyzed by the explicit analysis in a few seconds, thus using the combination makes an improvement on the run-time compared to using only predicate analysis configuration.

We can also add CBMC with $k = 1$ to the combinations since it only spends less than 8 s to run the entire set of benchmarks and the overhead of running CBMC will not effect the analysis. Also, it may be beneficial to run it since it can detect a bug in less than 1 s in some cases. Combination B, shown on the column *Comb. B* combines CBMC with loop bound $k = 1$, explicit analysis with 10 s time limit, and the predicate analysis with LBE. Since CBMC cannot analyze any of the programs from `systemc` benchmarks successfully, and since a majority of the total time spent is used for running `systemc` programs, the effect of it cannot be observed in the total run-time. The results show an improvement in the total run time for `locks` and `ntdrivers` benchmarks though. It is the expected behaviour since CBMC can verify or detect bugs for some of these programs. Adding CBMC to the process does not negatively effect the status of the analysis, so it is a good idea to combine it with other analyses.

Experiments that use predicate analysis in combinations is allowed to use a maximum Java heap size of 4 GB since we want to allocate memory for theorem provers and interpolation solvers. On the other hand, we do not need to make heavy time and memory consuming calls from the explicit analysis, so when only explicit analysis is used, the heap size is set to 12 GB. Since it is not possible to modify the allocated heap size in the middle of the analysis, the explicit analysis used in combinations also uses 4 GB of heap. It does not really make any significant difference but it might be one of the reasons of different behaviour for explicit analysis when combined with other analyses.

## 4.4 Discussion

All analysis techniques have different properties and in many cases, even combining them in parallel and using intermediate settings may not improve the efficiency of the process. Sequential composition provides an intuitive but practically useful environment for verification tasks at no cost. By combining several configurations, the effectiveness of the verification

process can be improved significantly, both in the number of verified programs and in the run-time. CBMC with a small loop bound is highly efficient in finding shallow bugs, and explicit analysis is good in finding possibly deep but still "easy" bugs. For proving program safety, predicate analysis is the best choice as a precise model checker. Thus, we first spend a limited effort in finding bugs with the efficient configurations. If no result is given, we switch to the next more powerful and more expensive analysis. So, via sequential composition framework, we first run an inexpensive shallow analysis to identify the easy bugs, then run a slightly more expensive, more in-depth analysis to solve more instances, and last run the most expensive analysis only for the remaining programs that were not yet successfully checked before. This can be achieved by providing configurations and it requires no user-interaction.

This design is orthogonal as the combined analyses can run independently and they will not effect each other. The main advantages of sequential composition is even more obvious when different analysis can communicate between each other. In the following chapters, we will introduce mechanisms that can be used for the communication and the results of one analysis can be used to improve the results of another analysis. Providing the sequential composition framework that handles the restarting task automatically, CPAchecker provides great flexibility to the user for optimizing the verification process.

# Chapter 5

# Verification Assumptions and Conditions

AssumptionCollecting

In this chapter, we introduce two concepts to improve presision of program analysis and approaches which try to minimize the number of UNKNOWN results for program analysis. As we have already discussed, software verification is an undecidable problem and the most common output for software analysis tools is still UNKNOWN. If the sources of imprecision are defined and used by the tool, UNKNOWN cases or crashes of the tool can be handled gracefully as we will demonstrate in this chapter. We reformulated the verification problem such that instead of returning an UNKNOWN answer, return an assumption as result. Running the verification under a set of assumptions also makes it possible to use the sequential composition framework that we introduced in Chapter 4 more efficiently. Even though the state space of two different techniques cannot be compatible with each other due to differences in their abstract domains, a common representation of outputs formulated as assumptions makes it possible to use the results of one analysis to be employed as an input for another analysis. In Chapter 4, we used CPAchecker with sequential composition only to restart the analysis with another approach or another tool after no final result is reported. In this chapter, we will use the sequential composition framework to restart the analysis to utilize the incomplete result from an analysis. We also use a related concept, verification conditions, to run program analysis. Verification conditions are forms of assumptions that limit the state space of the program using a specified condition. Verification conditions

guide the analysis based on statistics gathered during the analysis, and at the end of the analysis, CPAchecker reports how the verification conditions guided the analysis. We use this report as well within the sequential composition, update the conditions based on the previous runs of the analysis, and restart the analysis. At the end of the chapter, we present our experiments which demonstrate that using verification assumptions and conditions can effectively eliminate many cases of UNKNOWN results and increase the efficiency and precision of the analysis.

## 5.1 Motivation

As we discussed in previous chapters, formal verification tools aim to capture how the system will react with any possible input. This is a very costly operation, so it is impossible to check for total correctness of the program with formal methods. Several techniques have been developed that aim to verify partial correctness of the system such that instead of trying to prove total correctness, they can be configured to check some properties of the software.

Software verification tools aim to make the analysis both precise and fast [59]. In order to achieve this, they use techniques such as modeling, abstraction and they usually run the analysis under some assumptions. For example, most model checking tools ignore integer overflows and focus on assertion checking and reachability analysis as we have demonstrated with CPAchecker. In such a case, stating the assumptions explicitly might be more useful rather than ignoring the condition altogether. We have proposed a conditional verification environment that use explicitly stated assumptions or conditions that will run an imprecise analysis [19, 20]. The condition of imprecision will either be explicitly defined by the user or it will be reported at the end of the verification process.

We have implemented that framework using CPAchecker so that the approach can be flexibly used by any implemented CPA [19, 20, 27]. Experiments showed us that using assumptions for the verification can be used for more than one purpose. We list them in three categories as follows [19, 20]:

1. **Conditional Verification:** Providing a mechanism to the user for starting the verification using assumptions will be useful when the user wants to run a partial verification of the system for practicality reasons. This mechanism can be provided in two different ways: First, the user can specify a bound or condition that are satisfied by some

variables. For example, if an operation gets its input from an external program and the user knows that the input is always a positive integer, the analysis can be started with an assumption stating that the input is always greater than 0. Second, we provide a set of conditions that specify how the verification might proceed. Using these conditions helps the user to focus on some specific parts of the verification problem or to limit some resources. For example, the user might specify that a program location should be checked only for a limited number of times, so the analysis can spend time on analyzing different parts of the verification problem rather than traversing in a single loop.

2. **Assumption Generation:** Software model checking may not return an answer to the problem in some cases. There might be several reasons for a software model checking failure. The memory resources of the machine might not be sufficient to complete the task, the analysis might take an unreasonable amount of time to complete the analysis or the verification tool might not handle some parts of the code under verification (e.g. unhandled program statements). For all these cases, a significant amount of work is performed but since the model checker fails, no result is reported to the user.

   The verification framework we implemented can be used to report any work performed to the user in all cases. The conventional model checking process might report UNSAFE with a bug report, it might report that all states within the current model are checked and the verification specification holds, or stops with no results. With our proposed framework, the last case is replaced by a report to the user which is generated during the execution of the program. We integrated assumption generation with the conditional verification framework. If the analysis is stopped at some point because a starting assumption or an condition is used, this is also reported to the user. This mechanism makes it possible for the user to check for the main source of imprecision.

3. **Conditional Model Checking with Sequential Composition:** The two mechanisms explained above will produce a verification result for the given program even in cases where the conventional verification was failing. Reporting generated assumptions to the user will give some hints to increase the coverage of the analysis. In such a scenario, the user might use the same verification technique by changing assumptions and conditions, or can use another verification technique or tool. This is easily usable with the sequential composition framework. Our experiments have shown that even

the conditions providing basic resource limits is very successful at optimizing verification results when we use them to sequentially combine with analyses using different techniques and assumptions.

### 5.1.1 Examples

```
1  void main() {
2    int x = 1;
3    int y = 1;
4    if (nondet_int()) {
5      while (x < 10000) {
6        x++;
7      }
8      assert (x == 10000);
9    } else {
10     x = 0;
11   }
12   assert (y != 0);
13 }
```

Figure 5.1: Example program with loop

**Using conditions to improve verification results.** State space explosion is one of the main problems in software verification. This problem may manifest itself in a number of ways and it can be tackled by introducing imprecision to the analysis. For example, the existence of loops introduces additional challenges in model checking of software. Figure 5.1 shows a piece of code which demonstrates the problem on a loop with a large number of iterations. If predicate analysis is used to analyze the code shown in the figure, the analysis might fail to terminate with a meaningful result in reasonable time limits.

Since we use CEGAR for refining the precision of predicate analysis, each time the loop is unrolled, a new predicate will be added to the predicate list and a new abstraction will be computed after each iteration due to the assertion at line 8. This procedure will be repeated as many times the number of iterations (possibly infinitely many). If the loop has a large number of iterations as in the figure, the analysis will fail to terminate with a result. We can see that there is another assertion after the loop ends (at line 11) which is straightforward to check and the condition to be checked is independent of the loop. So, the user can specify a bound on the number of unwindings of the loop using a condition. In that case, the loop will be bounded and checked a limited number of times which is determined by the specified

limit. It will be an imprecise analysis because all the paths that might lead to an assertion failure at line 8 will not be analyzed, but another assertion will be checked instead of failing with no results and the source of the incompleteness will be reported. This will make it possible for the tool to isolate a part which is not analyzed throughly and the user can use another approach (such as testing) to check for any potential bugs on the part that have not been analyzed.

```
1  int main() {
2    int p;
3
4    if (p) {
5      int i;
6      for (i = 0; i < 1000000; i++);
7      assert(i >= 1000000);
8
9    } else {
10     int x = 5;
11     int y = 6;
12     int r = x * y;
13     assert(r >= x);
14   }
15   return(0);
16 }
```

Figure 5.2: Example program with non-linear safety condition (taken from [19, 20])

**Restarting analysis to improve verification results.** The program shown in Figure 5.2 presents a situation where the code has two parts that have different properties. In the first part (lines 4-8), there is a loop with a large number of iterations similar to the scenario in the first example. The second part (lines 9-14) contains statements that can be represented by non-linear relations. If we start analyzing this program with an initial predicate ($i \geq 1000000$), the first part of the program can be verified easily by predicate analysis. (i.e., the assertion at line 7 does not fail.)

Since predicate analysis uses SMT solvers to compute abstract states, its capabilities are limited by theories which are supported by the called SMT solver. Non-linear relations cannot be expressed in SMT solvers that solve decision problems in first-order logic. Therefore, the assignment `r = x*y` can be expressed as an uninterpreted function and the model checker will report UNSAFE for the assertion at line 13 which is a false alarm. In that case, we can use generated assumptions to verify different parts of the program with

different techniques. For the example given in Figure 5.2, we can start the analysis with the initial predicate and verify the first part of the program with predicate analysis. Then, we will start the analysis with an assumption to exclude the first part which is already verified and use an analysis that tracks the values of integer variables explicitly such as explicit analysis. So, the explicit analysis can easily verify the second part of the program. Note that, even when we have the initial predicate ($i \geq 1000000$), explicit analysis cannot use this information and it has to unwind the loop 1000000 times which will end up using all memory resources and this will crash the tool producing no results. So, using assumptions in combination with sequential composition can be combined to verify the given program in Figure 5.2 which we would not be able to analyze if we used predicate analysis or explicit analysis separately.

Figure 5.3 from [19] presents the ARG for a program which can be analyzed more efficiently using some conditions. This is a well-known problem for program analysis, and a similar case is demonstrated on the example given in Figure 5.1. The gray-shaded part contains abstract elements that have a depth larger than 8 on the ARG. So, from the initial abstract state, at least 7 abstract states are discovered to compute abstract states shown in this part. In some cases, there might be bugs which does not require to search the state space of the program deeply but rather can be found immediately if other parts of the state space is searched. Thus, limiting the search with a condition might help the analysis to complete a shallow state space search first. We will introduce a method that determines the limit for a condition automatically in this chapter.

### 5.1.2 Contributions and Applications

We have implemented and used verification assumptions and verification conditions in CPAchecker. Also, these mechanisms are used within the sequential composition framework and we monitored and evaluated potential usages of them.

- *Always Return Answer:* Using the verification assumptions, the model checking problem can be reformulated: Instead of failing to return any answers, summarize the work done by the tool and return the reached states with a set of assumptions.

- *Use Results for Sequential Composition:* An imprecise (or unfinished) analysis can return its output formulated as specified by conditional model checking. This result can be used by our sequential composition framework as an input to another tool or

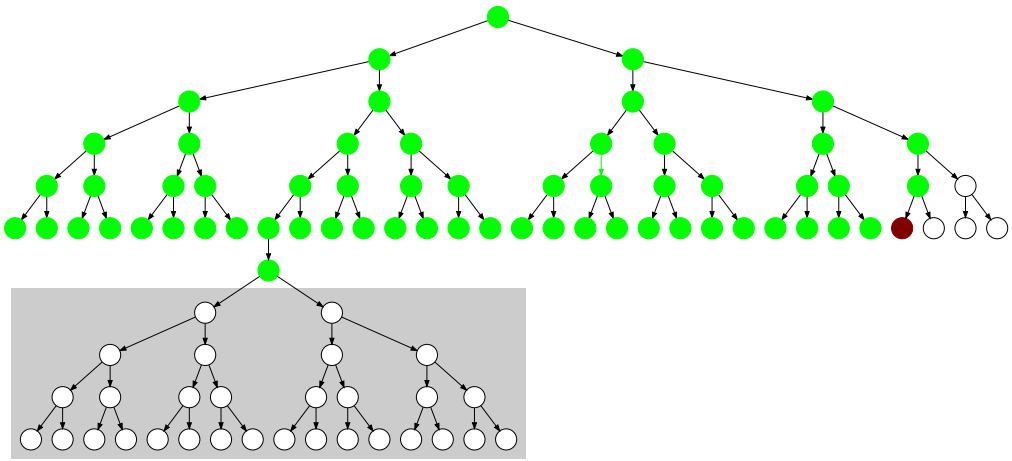

Figure 5.3: Example state space for a program which is analyzed using a limiting condition on the length of the path which is taken from [19]. Limiting the length of the path lets the analysis to skip shaded part and find the error state shown with the brown-colored node.

CPA. Using a common output-input format makes it possible for skipping the work already completed by the previous analysis and this may improve the efficiency of the second algorithm dramatically.

- *Partial Analysis:* Conditional verification makes it possible to limit exploration of the state space of the program. Using a verification assumption or condition as an input, the analysis will not try to generate all possible abstract states of the program but it will skip some parts which are specified by the input. Since the input and the output of a partial verification process is defined by a set of assumptions, the result will be reported as an imprecise result and the user will be able to check if there are any parts of the program which were skipped or analyzed under an assumption.

- *Improved Bug Hunting:* Running the analysis with conditions will guide it to search for particular states of the ARG as shown in the examples. This might help the analysis to reach to an UNSAFE condition faster if irrelevant parts of the state space is skipped. Using conditional model checking with sequential composition makes it possible to generate a condition that will determine relevant parts effectively. This approach improves the bug hunting significantly in the existence of a real bug in the program.

- *Improved Coverage:* Even in cases where a program does not contain any bugs, the conditional verification might improve the coverage of the analysis. If the analysis spends a significant time on discovering a specific part of the program, this might consume significant amount of memory and time resources. Had this part been skipped upon detecting that it consumes too much resources, the analysis may start searching different paths of the state space and cover a larger part of the state space.

- *Understanding the Behaviour and Generating Benchmarks:* Guiding the model checking using conditional model checking serves as a mechanism for monitoring the analysis. The statistics and generated assumptions at the end of the analysis might be very useful for understanding the behaviour of the analysis and structure of the code. This also makes it possible to insert some conditions to the program that adhere to some specific properties. For example, the user might want to create a benchmark program which contains a bug within some limits of the state space of the program. Using conditions to satisfy these properties, the user can evaluate the output of the program and create such a program.

- *Compatibility with any CPA:* We implemented analyses as CPAs that use and create verification assumptions and conditions. That way, they can be composed with any other analysis which is encoded as a CPA using CPAchecker. Therefore, conditional model checking is not tightly coupled with a specific algorithm but it is compatible with any other algorithm in CPAchecker. Running as a separate component makes conditional model checking highly configurable and flexible.

## 5.2 Verification Assumptions

A verification assumption is a state predicate that simply defines under what assumption a state is reached. For example, if a state for a location $l$ which can be represented by the formula $\varphi$, an assumption $\psi$ for that location and that state indicates that the state $\varphi$ for location $l$ is computed when we assume that $\psi$ is *true* at location $l$. A straightforward application of using assumptions would be to run an analysis with a set of assumptions as input. That approach would make it possible for the user to specify some bounds for a variable for example, without modifying the source code or inserting explicit assume

```
1  int main(void)
2  {
3    int value1;
4    int value2;
5    int max;
6
7    value1 = randFunc1();
8    value2 = randFunc2();
9
10   if(value1 > value2){
11      max = value1;
12   }
13
14   else if(value2 > value1){
15      max = value2;
16   }
17
18   if(max != value1){
19     if(max != value2){
20       goto ERROR;
21     }
22   }
23
24   END:
25   return (0);
26   ERROR:
27   return (-1);
28 }
```

Figure 5.4: Example C program

statements, or without modifying the analysis code itself. Let us revisit our example from Chapter 2 which is given in Figure 5.4 to remind.

If the user knows a bit more about the return values of random generator functions called at lines 7 and 8, the analysis might be more precise. For example, let us assume that the specification of `randFunc1()` states that the return value of the function is between 1000 and 10000. Furthermore, let us assume that we know the return value of `randFunc2()` to be an integer between 0 and 100. Remember that the edge labeled with `value1 = randFunc1();` was from CFA location 5 to CFA location 6, and the edge labeled with `value2 = randFunc2();` was from CFA location 6 to CFA location 7. Now we can specify assumptions as $\varphi_1 : (l = 6) \Rightarrow ((value1 > 1000) \wedge (value1 < 10000))$ and $\varphi_2 : (l = 7) \Rightarrow ((value2 > 0) \wedge (value2 < 100))$ and start the analysis using the assumption $\varphi_1 \wedge \varphi_2$.

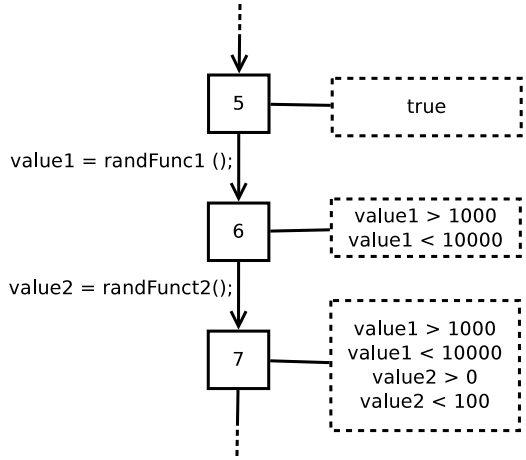


Figure 5.5: ARG nodes for locations 5, 6 and 7 for the example given in Figure 5.4 when the analysis starts with using the assumptions $(l = 6) \Rightarrow ((value1 > 1000) \wedge (value1 < 10000))$ and $(l = 7) \Rightarrow ((value2 > 0) \wedge (value2 < 100))$

Figure 5.5 shows the relevant fragment of the ARG for the example. For the sake of simplicity, we assume that we use a numerical domain that keeps bounds of variables for that example. The nodes labeled with CFA locations are ARG nodes and dashed boxes associated with each ARG node shows the abstract state computed for that particular node. For location $l = 5$, there is no assignment to variables and the abstract state has no bounds for variables. From location 5 to 6, there is an assignment to the variable `value1` from an external call. Thus, the analysis cannot decide what bounds shall be set for the variable. Since we started the analysis using the assumption $\varphi_1$, the analysis uses this assumption and sets the bounds for `value1` on location 6. The other assumption $\varphi_2$ will be used for location 7 in the same manner and the abstract state will be updated accordingly. If we have that specific information about the return values of the two random functions, we can use assumptions to rule out a false alarm in the program since `value1` will always be larger than `value2`. The result of the analysis will even show to the user that `else if (value2 > value1)` condition always returns *false*. That makes line 15 of the code shown in Figure 5.4 unreachable (dead code).

### 5.2.1 Conditional Model Checking and Assumptions as Output

In addition to their direct applications as an input to software analysis, verification assumptions can be used as an output of the analysis. The most common output for model checking (or software analysis in general) is UNKNOWN instead of a SAFE or UNSAFE answer. If the result is SAFE or UNSAFE, the output can be certified by a proof or a counter-example if the safety specification is violated [16, 69, 96]. There are several reasons of imprecision of an analysis: The state space of the program could be very large that the resources (time or memory) are not sufficient to run the analysis and the analysis crashes, the tool does not handle some operations that the program uses, or the analysis is not precise enough to capture some facts about the behaviour of the program.

We formalized the model checking problem using assumptions to eliminate UNKNOWN answers to the verification problem. According to this; the analysis will output an assumption with the result even in cases where the result is UNKNOWN to show where the analysis fails to be sound. Note that there might be several assumptions for different locations of the program as in the example presented in Figure 5.5. We can always take the conjunction of these assumptions and represent them with a single formula. So, when we refer to a set of assumptions or an assumption, the difference is syntactic and the choice is merely to make the representation easier. We represent the assumption with $\Psi$. In the reformulated model checking problem; (i) If the result of the analysis was SAFE, the output will be SAFE with $\Psi = \mathit{true}$, (ii) If the result of the analysis was UNSAFE, the output will be UNSAFE with $\Psi$ that satisfies the specification, and (iii) If the result of the analysis was UNKNOWN, the output assumption $\Psi$ will summarize the work completed by the analysis. In other words, the resulting state space of the program satisfies the assumption $\Psi$. That description applies to the analysis when we use assumptions as input for the program as well. What we introduce here is to run the analysis and instead of terminating the analysis with no results in cases such as time-outs or crashes due to memory limits, to finish the analysis and summarize the work done using an assumption. Later in this chapter, we will introduce verification conditions that can be used to avoid failures of the analysis and to terminate all analyses with an output.

**Definition** $\Psi$ *Soundness:* We say that an analysis is sound under assumption $\Psi$ if the state space of the program is verified under the assumption $\Psi$. A program is $\Psi$-sound if the CPA algorithm terminates after computing the state space that satisfies $\Psi$ due to Theorem 2.3.1.

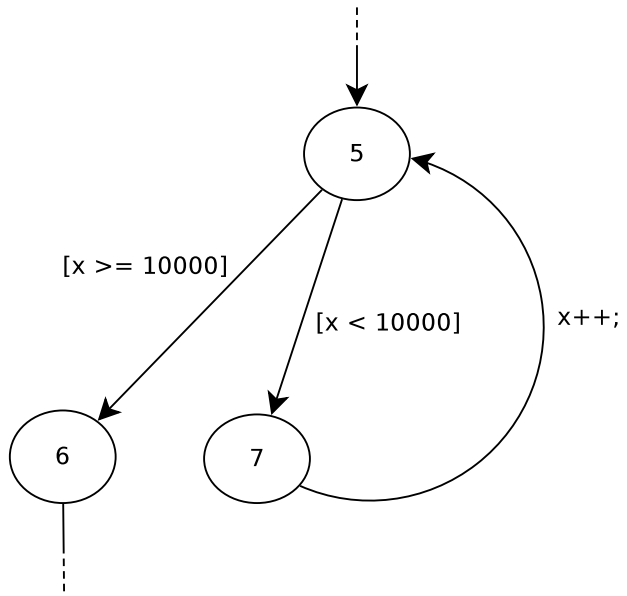


Figure 5.6: A part of the CFA for the example program with loop given in Figure 5.1

$\Psi = true$ is the weakest assumption and it is the traditional setting for most model checking problems. We are interested in analyses which produce the weakest possible assumption, preferably $\Psi = true$, but if the analysis cannot be finished with that assumption, we want to terminate with a stronger assumption.

The example shown in Figure 5.1 demonstrates a case where we can summarize the uncompleted verification process using an assumptions. If there is a limit for unwinding the loop as explained for this example, the analysis will stop unrolling the loop after that limit is met. This suggests that the analysis will not continue on this path after the loop is unrolled for the last time, but since discovering a section of the state space is skipped, the analysis result will be imprecise. Let us assume that the loop will be unrolled 10 times and then the exploration of the successor states on this path will be stopped. A part of the CFA for this program which models the loop is shown in Figure 5.6.

Figure 5.7 shows the resulting ARG for the part of CFA shown in Figure 5.6 using the explicit analysis. The ARG in the figure shows the $8^{th}$, $9^{th}$ and $10^{th}$ unrollings of the loop. The value of the variable `y` does not change through the execution of the loop and the values of `x` is incremented and assigned to a new value when the edge from location 5 to location 7

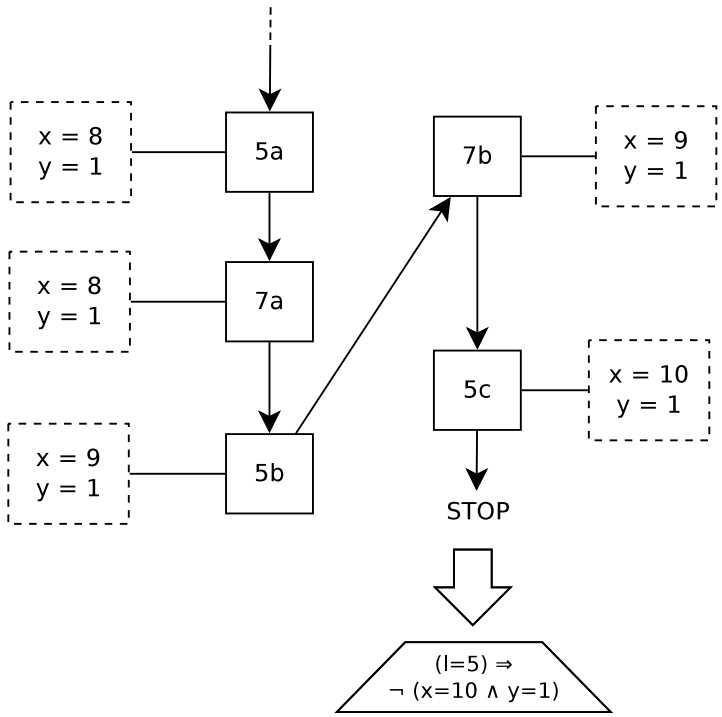


Figure 5.7: A part of the ARG for the example program with loop given in Figure 5.1

is proceeded. The dashed boxes show the abstract states for program locations that are inside the loop. If we have a way to guarantee that a loop will be unrolled a limited of times by the analysis, we can create a condition which forces the tool to stop building the ARG after the $10^{th}$ unrolling (later we will introduce verification conditions to create and use similar conditions). After the node labeled with `5c` is computed, the location with node id 5 is visited 10 times and we do not want to continue computing the abstract states on this path anymore and force the analysis to stop for the path. The analysis may continue on a different path but since we skipped a part of the ARG, the analysis will not be sound. So, we create a verification assumption $\psi : (l = 5) \Rightarrow \neg((x = 10) \wedge (y = 1))$ shown in the trapezoid. The assumption $\psi$ is reported at the end of the analysis with the reached states and implies that the location with id 5 is excluded from the analysis if the abstract state was $((x = 10) \wedge (y = 1))$ at this point. Thus, the state $((x = 10) \wedge (y = 1))$ should either be unreachable at location 5 or it is excluded from the analysis.

### 5.2.2 Formalization and Implementation of Assumptions

We have implemented verification assumptions as a component in CPAchecker. In order to make it compatible and usable with other analyses, we implemented an *assumptions storage CPA* and used it with an *assumption collecting algorithm*. We also introduced a CPA for explicitly stating unsoundness due to overflows in the analysis.

**CPA for Assumptions Storage.** The *assumptions storage CPA* is used to generate and store assumptions. The CPA $\mathbb{A} = (D_{\mathbb{A}}, \Pi_{\mathbb{A}}, \rightsquigarrow_{\mathbb{A}}, \mathsf{merge}_{\mathbb{A}}, \mathsf{stop}_{\mathbb{A}}, \mathsf{prec}_{\mathbb{A}})$ consists of the following components: The abstract domain $D_{\mathbb{A}}$, the set $\Pi$ of precisions, the transfer relation $\rightsquigarrow_{\mathbb{A}}$, the merge operator $\mathsf{merge}_{\mathbb{A}}$, the stop operator $\mathsf{stop}_{\mathbb{A}}$ and the precision adjustment operator $\mathsf{prec}_{\mathbb{A}}$.

1. The domain $D_{\mathbb{A}} = (C, \mathcal{E}_{\mathbb{A}}, [\![\cdot]\!]_{\mathbb{A}})$ is based on the semi-lattice $\mathcal{E}_{\mathbb{A}} = (\mathcal{F}, \mathit{true}, \mathit{false}, \sqsubseteq, \sqcup)$. $\mathcal{F}$ is the set of quantifier-free formulas over variables from $X$. $\top = \mathit{true}$ is the weakest assumption which stands for the top element which leaves a state to be computed under no assumptions. $\bot = \mathit{false}$ is the bottom element. $\sqsubseteq$ is modeled by the implication relation, $\Rightarrow$ over formulas in $\mathcal{F}$. $\sqcup$ is defined by the logical conjunction operator $\wedge$. An abstract state represents an assumption made by the model checker about the concrete states of the program. So, $[\![a]\!]_{\mathbb{A}} = \{c \in C \mid c \models a\}$ where $a$ is a formula from $\mathcal{F}$. The assumption can be specified by the user as well, and a specified assumption is converted to an element represented in $\mathcal{F}$.

2. The set $\Pi_{\mathbb{A}}$ contains only a single value: $\Pi_{\mathbb{A}} = \{\pi_{\mathbb{A}}\}$.

3. The transfer relation $\rightsquigarrow_{\mathbb{A}}$ is the set $\{(e, g, e', \pi) \mid e \neq \mathit{false},\ g \in G,\ e' = \mathit{true},\ \pi = \pi_{\mathbb{A}}\}$. The successor state is always the abstract state that makes no assumptions. This will be refined by the strengthen operator $\downarrow$ of the composite CPA. If the predecessor state has the assumption *false*, the analysis does not proceed because the assumptions already exclude the current path.

4. The merge operator computes the conjunction of both assumptions: $\mathsf{merge}_{\mathbb{A}}(e, e', \pi) = e \wedge e'$.

5. The termination stop considers abstract states individually: $\mathsf{stop}_{\mathbb{A}}(e, R, \pi) = (\exists e' \in R : e' \Rightarrow e)$ It returns true, if there is a state in the set of reached states that has a stricter assumption.

6. The precision adjustment function does not change the element and the precision: $\mathsf{prec}_{\mathbb{A}}(e, \pi, R) = (e, \pi)$

**CPA for Overflow Monitoring.** We introduce a CPA which can be used to model an assumption that is specified by the user in [19, 20] as given in this section. Variables of a computer program are always bounded (e.g., to 32 bits) and therefore have a minimum as well as a maximum value. If the value exceeds the maximum, an overflow occurs. However, predicate analyses usually ignore overflows and model variables as unbounded mathematical integers due to limitations of expressibility in linear arithmetics. Ignoring this property would make the analysis imprecise.

We defined a CPA that monitors the analysis. Then it generates the necessary assumptions under which the analysis is sound. These assumptions can also be used to remove infeasible paths from the the explored state space. We implemented this CPA on CPAchecker.

The *CPA for Overflow Monitoring* tracks the assumptions for overflows of program variables. For simplicity, we assume that all program variables have the same type with a minimal value of $\mathsf{MIN}$ and a maximum value of $\mathsf{MAX}$.

1. The domain $D_{\mathbb{O}}$ is based on boolean formulas: $D_{\mathbb{O}} = (C, \mathcal{E}_{\mathbb{O}}, [\![\cdot]\!])$, with $\mathcal{E}_{\mathbb{O}} = (\mathcal{F}, \Rightarrow)$, where $\mathcal{F}$ is the set of quantifier-free boolean formulas over variables from $X$.

2. The set of precisions contains only one precision: $\Pi_{\mathbb{O}} = \{\pi_{\mathbb{O}}\}$.

3. The transfer relation creates an assumption if the current program transition might lead to an overflow. If the edge $g$ is an assignment of the form $x = t$, where $x \in X$ and $t$ is a term over variables from $X$, the successor state $\psi'$ is the formula $(x \geq \mathsf{MIN}) \wedge (x \leq \mathsf{MAX})$. In other cases, the successor state is *true*. We assume that overflows occur only at the end of the computation, when the result of $t$ is assigned to $x$. Statements in which overflows occur in intermediate results are transformed into simpler statements by pre-processing.

   If this CPA produces an assumption, it will be added to the abstract state of the CPA for predicate analysis by the strengthen operator $\downarrow$. This will help the analysis by excluding parts of the state space that are infeasible due to overflow for a variable.

4. The merge operator combines elements by taking the conjunction: $\mathsf{merge}_{\mathbb{O}}(\psi, \psi', \pi) = \psi \wedge \psi'$

5. The termination stop $\mathsf{stop}_{\mathbb{O}}$ always returns *true*.

6. The precision adjustment function never changes anything: $\mathsf{prec}_{\mathbb{O}}(\psi, \pi, R) = (\psi, \pi)$

We will introduce how the strengthen operator $\downarrow$ is used later when defining the *composite CPA for assumptions*.

**Assumption Collecting Algorithm.** We present the *assumption collecting algorithm* in Algorithm 4. After the analysis terminates, the output is processed to report the assumption along with the result of the analysis. The *assumption collecting algorithm* runs a CPA algorithm and the user should provide the CPA algorithm and a CPA composed with the *assumptions storage CPA*. In Algorithm 4, we assume that we use a composite CPA composed of three CPAs; the location analysis CPA $\mathbb{L}$, the assumptions storage CPA $\mathbb{A}$ and a CPA $\mathbb{T}$ which has domain elements that can be converted to a first-order formula, such as predicate analysis CPA.

**Algorithm 4** *AssumptionCollecting*$(A, \mathbb{C}, R_0, W_0)$

**Input:** a CPA algorithm $A$
a CPA $\mathbb{C} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ with the *assumptions storage CPA* and
a leaf CPA $\mathbb{C}_{\mathbb{T}}$,
a set $R_0 \subseteq E \times \Pi$ of abstract states with precision,
a subset $W_0 \subseteq R_0$ of frontier abstract states with precision,
where $E$ denotes the set of elements of the semi-lattice of $D$
**Output:** a set of reachable abstract states with precision,
assumption generated for the analysis
**Variables:** two sets reached and waitlist of elements of $E \times \Pi$ and a formula assumption

```
assumption := true;
(reached, waitlist) := A.run(R_0, W_0);
for each (e, π) ∈ reached do
  e := (l_i, e_A, e_T)
  if e is ERROR element then
    assumption := assumption ∧ (l = l_i ⇒ ¬e_T);
  if e ∈ waitlist then
    assumption := assumption ∧ (l = l_i ⇒ ¬e_T);
  else
    assumption := assumption ∧ (l = l_i ⇒ (¬e_T ∨ e_A));
return (reached, assumption)
```

The algorithm updates a variable to report the generated assumptions, assumption, as it traverses the sets reached and waitlist. The variable assumption is a formula and it is set to

*true* at the beginning of the algorithm. After the CPA algorithm returns the sets reached and waitlist as its outputs, the algorithm starts iterating over elements $(e, \pi)$ of reached. Note that since we use three CPAs, the abstract element $e$ is composed of three elements: the location element $l_i$, the assumption element $e_{\mathbb{A}}$ and the element for the other CPA $e_{\mathbb{T}}$. For example, $e_{\mathbb{T}}$ might be a region over predicates for predicate analysis CPA. If there are error elements in the set reached, the assumption $(l = l_i \Rightarrow \neg e_{\mathbb{T}})$ is generated for these elements. A subset of the error elements might be on a verified error path, or an error path determined to be unfeasible. If the error element is genuine, then the analysis will report a condition that encodes the error path. If they are found to be unfeasible (for example by CBMC verification) but are still in reached, then the analysis might be imprecise as for the case we explained for the *explicit analysis CPA* with CBMC. Adding $(l = l_i \Rightarrow \neg e_{\mathbb{T}})$ to the assumption will model the unsoundness by stating that the analysis was stopped at location $l_i$ when the state was $e_{\mathbb{T}}$ and the path was not analyzed further. For elements which are also in waitlist, a similar approach is taken. Since these elements were generated but not processed yet, they introduce unsoundness and the assumption $(l = l_i \Rightarrow \neg e_{\mathbb{T}})$ is generated for them. For all other elements in reached, the assumption $(l = l_i \Rightarrow (\neg e_{\mathbb{T}} \vee e_{\mathbb{A}}))$ is created which states that for all abstract states, the region satisfies the assumption at that program location. The resulting assumption will be the conjunction of all the assumptions created after iterating through elements of reached.

This approach can be applied to processes like predicate refinement. If the analysis detects an infeasible error path $(e_0, \ldots, e_n)$ but fails to compute a better precision that would exclude this path, the assumption $(l = l_i) \Rightarrow \neg e_{\mathbb{T}_i}$ is added for all states $e_i = (l_i, \cdot, e_{\mathbb{T}_i})$ beginning with the first unreachable state in the path until $e_n$.

Figure 5.8 shows the architecture of the assumption collecting algorithm. The assumption collecting algorithm acts like a wrapper around the CPA algorithm. The composite CPA consists of a *location CPA*, an *assumptions storage CPA* and a leaf CPA which is shown as $\mathbb{C}_{\mathbb{T}}$ in Algorithm 4. The leaf CPA can be a composite CPA as well. The leaf CPA should implement the *formula reporting interface* for converting its abstract element to a formula. For example, for the *predicate analysis CPA* element, the formula that represents the region itself is reported. The *explicit analysis CPA* keeps the abstract state as a map from variables to integer values. So, the abstract element for explicit analysis should be converted to a formula in order to include it in the resulting assumption. The formula representation for an explicit analysis element $e_{\mathbb{E}}$ is $\varphi_{e_{\mathbb{E}}} = (\bigwedge_{x \in X} x = e_{\mathbb{E}}(x))$. For example, if an explicit analysis

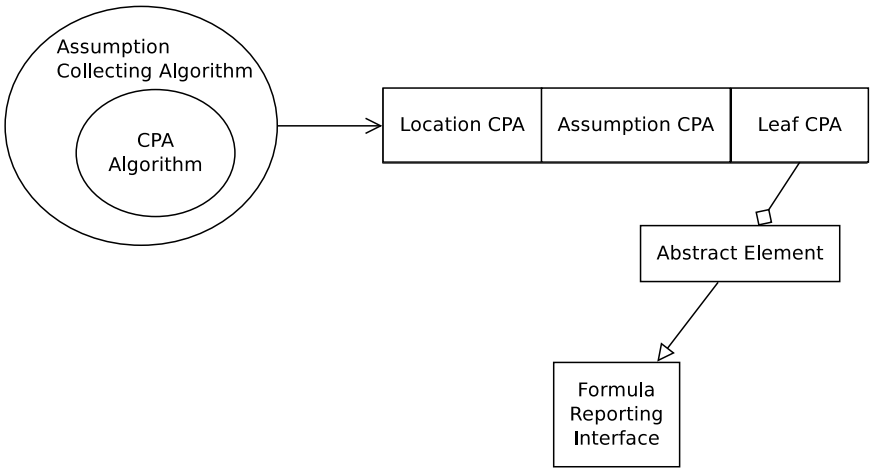


Figure 5.8: Architecture overview of assumption collecting algorithm

element $e_{\mathbb{E}}$ maps two variables from $X$ to their integer values, $x_1 = c_1$ and $x_2 = c_2$, then the resulting formula to represent this state is $\varphi_{e_{\mathbb{E}}} = ((x_1 = c_1) \wedge (x_2 = c_2))$. The assumption collecting algorithm retrieves the state representing formula from the methods provided by the *formula reporting interface*. If an element does not implement this interface, *true* is returned. We have implemented and used formula reporting methods for predicate analysis and explicit analysis in CPAchecker.

## 5.3 Verification Conditions

In many cases where the analysis fails, the reason is usually consuming all resources or timing-out as the state space gets very big or the abstraction computation is very expensive. The user might not know at what point or when the analysis will fail. In this section, we introduce verification conditions that can be used to terminate the analysis or limit the state space search using some pre-defined restrictions on the analysis. We implemented several techniques to prevent the analysis tool from failure. We have used these techniques with verification assumptions and sequential composition to make them practically useful instead of producing imprecise results.

First, we monitor the progress for critical components of the model checker such as the abstraction computation or refinement, or the size of the state space explored. When a specific component fails to terminate within the given limits (exhausts its assigned time or memory limits), or when the analysis fails to return an answer, the monitor detects the problem, terminates the component or the analysis and generates an assumption that excludes the corresponding states from the verification result.

Second, we implemented several verification conditions that try to predict situations in which the model checker should not continue exploring a path on the ARG. In such a case, the tool generates assumptions excluding that part and continue with exploring another part of the state space. Such conditions make it possible to verify larger parts of the program, instead of spending all time on a particular, unsolvable problem. Thus, if we cannot completely verify a program, we at least obtain a verification coverage that is as large as possible. Our experiments show that these conditions can also significantly improve the performance; for example, programs for which the model checker previously failed after spending minutes, can now be proved to violate the specification within seconds.

The example given in Figure 5.1 in Section 5.1.1 presents a case where the model checker skips a part that is unsolvable and work on another section to detect a safety violation. On this example, as explained in Figure 5.7, the number of unwindings of the loop is limited. We presented verification conditions implemented in CPA framework using CPAchecker. This makes it possible to use verification conditions with any analysis that is implemented as a CPA. For example, the example to limit the number of unwindings of the loop is applicable to any CPA using a CPA to monitor the locations on a path and to stop the analysis when the specified threshold is exceed for a location.

### 5.3.1 Failure Classes and Failure-preventing Conditions

We present a list of verification conditions that is classified based on what they monitor and how they guide the execution. This list is taken from [19].

**Classes of Failures.** There are various reasons for a model checker to fail during the verification process. We are particularly interested in classifying the failures according to the part of the verification algorithm where the problem occurs.

1. **Global Progress.** If the model checker does not stop continuously adding new abstract states to the set of reachable states, it will sooner or later run out of resources.

For example, in an explicit-value analysis, a loop might be unfolded many times (cycling) and many explicit values are stored for a particular location. To prevent the model checker from running into such a situation, we can measure the number of abstract states in the reached set for a certain location and generate an excluding assumption. Similarly, we can monitor the total number of abstract states, total memory consumption and total time consumption.

2. **Post Computation.** The computation of the abstract successor might fail because the new abstract state might be too large, or too difficult to compute (memory, time). If this occurs, the monitoring component stops the analysis of the specific part, adds an excluding assumption, and lets the analysis continue on another path. Various conditions can be used to prevent the model checker from spending time on difficult program parts. For example, we could measure the size of the analysis precision (e.g., the number of predicates for a certain location exceeds a given threshold, which makes in turn the computation of the predicate analysis expensive) and stop exploring the current successor if a certain threshold is exceeded.

3. **Counterexample Analysis.** If an abstract error path has been found, it needs to be analyzed for feasibility and possible refinements. This step might fail in a predicate-abstraction-based analysis if the feasibility check of the path (performed by an external SMT solver) fails, or if the refinement process did not succeed in eliminating the infeasible error path (e.g., the used predicate-extraction technique might be not strong enough). If a failure of a component occurred, an assumption formula is generated for a pivot node on the path. To prevent the refinement component from failure, we can measure the length of the path, the predicates involved, the structure of the path formulas, etc.

**Preventing Conditions.** We have experimented with a number of verification conditions to prevent failure and to guide the state-space search to achieve a better verification coverage. We classify the conditions into the above-mentioned categories according to the location of occurrence in the analysis algorithm, and provide an incomplete list of example conditions (as an overview, cf. Fig. 5.9):

1. *Global Progress — Total Time and Space*: Measure the total time and memory consumption of the model checker. If a resource limit is reached, then the analysis is

| Component | Name | Conditions | Impl. |
|---|---|---|---|
| Global Progress | Total Time | $time$ | ✓ |
| | Total Space | $mem$ | ✓ |
| | # Abstract States | $\lvert\mathsf{reached}\rvert$ | ✓ |
| | # Abstract States per Loc. | $\#(\mathsf{reached}, loc)$ | |
| | Busy Edge | $\#(edge)$ | ✓ |
| Post Computation | Time for Post | $time(\rightsquigarrow)$ | ✓ |
| | Space for Post | $mem(\rightsquigarrow)$ | |
| | Size of State | $mem(state)$ | |
| | Path Length | $length(path)$ | ✓ |
| | Time Spent in Path | $time(path)$ | ✓ |
| | Repeating Locs. in Path | $\#(path, loc)$ | ✓ |
| | Assume Edges in Path | $assumes(path)$ | ✓ |
| CEX Analysis | Time for Refinement | $time(ref)$ | ✓ |
| | Space for Refinement | $mem(ref)$ | |
| | Size of Path Formula | $mem(pf)$ | ✓ |

Figure 5.9: Example Conditions for Conditional Model Checking; column *Impl.* lists the conditions we implemented

stopped and excluding assumptions are generated for all abstract states that are still in the waitlist at this moment. In contrast to 'hard' resource limits that we can set (e.g., with `ulimit`), these 'soft' limit enables the model checker to perform post-processing and output the verification results as an assumption formula.

2. *Global Progress — Number of Abstract States*: Measures the total number of the elements in the set of reached abstract states. If the number of states is larger than a certain threshold, the analysis is stopped and excluding assumptions are generated for all abstract states that are still in the waitlist at this moment.

3. *Global Progress — Number of Abstract States per Location*: Measures the number of elements in the set of reached abstract states that belong to each program location. If any location has more states attached to it than the limit, all post computations for incoming edges of this location are not performed anymore, and instead excluding assumptions are added.

4. *Global Progress — Busy Edge*: Measures the total number of post operations that the currently considered edge of the control-flow automata was involved in. If the edge is

used more often than allowed by a given threshold, post operations for this edge are not performed anymore, but rather excluding assumption are computed.

5. *Post Computation — Time and Space for Post*: Measures the time of each transfer in a component analysis. If the operation takes longer than the specified time limit or exceeds the specified memory limit, the operation is terminated, and an excluding assumption for the abstract state is added.

6. *Post Computation — Size of Abstract State*: Measures the size of the current abstract state, and if the specified threshold is reached, an excluding assumption is added. If predicate analysis is used, we take the number of disjuncts in the abstraction formula of the abstract predicate state.

7. *Post Computation — Path Length*: Measures the number of abstract states on the path to the current abstract state, and if the specified threshold is reached, an excluding assumption is added. (If several paths lead to the current abstract state, the maximum value is considered.)

8. *Post Computation — Time Spent in Path*: Measures the total amount of time consumed on the path to the current abstract state, and if the specified time limit is exceed, an excluding assumption is added. (If several paths lead to the current abstract state, the maximum value is considered.)

9. *Post Computation — Repeating Locations in Path*: Measures the number of occurrences of the current program location on the path from the root to the current abstract state in the abstract reachability graph. If a program location is encountered more than a specified number of times on a single path, an excluding assumption is added.

10. *Post Computation — Assume Edges in Path*: Measures the total number of assume edges seen on the path to the current abstract state, and if the specified threshold is exceed, an excluding assumption is added. (If several paths lead to the current abstract state, the maximum value is considered.)

11. *Counterexample Analysis — Time and Space for Refinement*: The execution of each refinement step for a given analysis is monitored. If the operation takes longer than the specified time limit or exceeds the specified memory limit, the operation is terminated, and an excluding assumption for the abstract state is added.

12. *Counterexample Analysis — Size of Path Formula*: We measure the size of the path formula, which is used for checking feasibility and for computing interpolants for predicate refinement. If the formula exceeds a certain size, we do not start the SMT solver on that formula, but rather add an excluding assumption.

### 5.3.2 Formalization and Implementation

We present CPAs for three conditions that we used in our experiments.

**CPA for Path Length on Path Monitoring.** We present a CPA that monitors the number of abstract states computed on a path and stops the analysis when this number exceeds a threshold. The *CPA for path length monitoring* $\mathbb{PL} = (D_{\mathbb{PL}}, \Pi_{\mathbb{PL}}, \rightsquigarrow_{\mathbb{PL}}, \mathsf{merge}_{\mathbb{PL}}, \mathsf{stop}_{\mathbb{PL}}, \mathsf{prec}_{\mathbb{PL}})$ tracks the number of abstract states computed on a path leading to the current abstract state, and stops the analysis when the threshold is exceed.

1. The domain $D_{\mathbb{PL}}$ is an integer to determine the length of the path: $D_{\mathbb{PL}} = (C, \mathcal{E}_{\mathbb{PL}}, [\![\cdot]\!])$, with $\mathcal{E}_{\mathbb{PL}} = ((\mathbb{N} \times \mathbb{B}), \sqsubseteq)$, $(e, t) \sqsubseteq (e', t')$ if $t = t'$ and $e = e'$.

   An abstract state consists of (1) an integer that stores how many abstract states have been computed on the path to the current state, and (2) a flag that shows whether number of abstract states computed on the path is larger than the user-specified threshold $k_{\mathbb{PL}}$ for the length of a path.

2. The set $\Pi_{\mathbb{PL}}$ of precisions contains only one precision $\pi_{\mathbb{PL}}$.

3. The transfer relation $\rightsquigarrow_{\mathbb{PL}}$ has the transfer $(e, t) \overset{g}{\rightsquigarrow}_{\mathbb{PL}} ((e', t), \pi)$ if (1) $t' = (e' > k_{\mathbb{PL}})$ and (2) $e' = (e + 1)$.

4. The merge operator combines elements by taking the maximum of the path lengths when control flow meets: $\mathsf{merge}_{\mathbb{PL}}((e, t), (e', t'), \pi) = (e'', t'')$ such that we have $e'' = \max(e, e')$ and $t'' = t \vee t''$.

5. The termination check always returns *true* because the decision whether an abstract state is covered does not depend on the state of the conditions CPA.

6. The precision is never changed: $\mathsf{prec}_{\mathbb{PL}}(e, \pi, R) = (e, \pi)$.

**CPA for Repeating Locations on Path Monitoring.** We present a CPA that monitors how often a single location was encountered on a path and stops the analysis when this number exceeds a threshold. The *CPA for repeating locations monitoring* $\mathbb{R} = (D_{\mathbb{R}}, \Pi_{\mathbb{R}}, \rightsquigarrow_{\mathbb{R}}, \mathsf{merge}_{\mathbb{R}}, \mathsf{stop}_{\mathbb{R}}, \mathsf{prec}_{\mathbb{R}})$ tracks the number of times a location has been seen on a path leading to the current abstract state, and stops the analysis when the threshold is exceed.

1. The domain $D_{\mathbb{R}}$ is based on a map from locations to natural numbers: $D_{\mathbb{R}} = (C, \mathcal{E}_{\mathbb{R}}, [\![\cdot]\!])$, with $\mathcal{E}_{\mathbb{R}} = (((L \to \mathbb{N}) \times \mathbb{B}), \sqsubseteq)$, $(e, t) \sqsubseteq (e', t')$ if $t = t'$ and for all $l_e$ in $L$, $e(l_e) = e'(l_e)$.

   An abstract state consists of (1) a map from locations to natural numbers that stores how often a location has been seen on the path to the current state, and (2) a flag that shows whether at least one location was seen more often than the user-specified threshold $k_{\mathbb{R}}$ for repetition of locations in a path.

2. The set $\Pi_{\mathbb{R}}$ of precisions contains only one precision $\pi_{\mathbb{R}}$.

3. The transfer relation $\rightsquigarrow_{\mathbb{R}}$ has the transfer $(e, t) \overset{g}{\rightsquigarrow}_{\mathbb{R}} ((e', t), \pi)$ for $g = (l, \cdot, l_s)$ if (1) $t' = (\exists l'' \in L : e'(l'') > k)$ and (2) the following holds:

$$\forall l'' \in L : e'(l'') = \begin{cases} e(l'') + 1 & \text{if } l'' = l_s \\ e(l'') & \text{else} \end{cases}$$

4. The merge operator combines elements by taking the maximum number of occurrences for each location when control flow meets: $\mathsf{merge}_{\mathbb{R}}((e, t), (e', t'), \pi) = (e'', t'')$ such that for all $l$ in $L$, we have $e''(l) = \max(e(l), e'(l))$ and $t'' = t \vee t''$.

5. The termination check always returns *true* because the decision whether an abstract state is covered does not depend on the state of the conditions CPA.

6. The precision is never changed: $\mathsf{prec}_{\mathbb{R}}(e, \pi, R) = (e, \pi)$.

**CPA for Assume Edges on Path Monitoring.** We present a CPA that monitors the number of abstract states for a conditional statement that are computed on a path and stops the analysis when this number exceeds a threshold. The *CPA for assume edges monitoring* $\mathbb{AE} = (D_{\mathbb{AE}}, \Pi_{\mathbb{AE}}, \rightsquigarrow_{\mathbb{AE}}, \mathsf{merge}_{\mathbb{AE}}, \mathsf{stop}_{\mathbb{AE}}, \mathsf{prec}_{\mathbb{AE}})$ tracks the number abstract states computed on a path for an assume edge leading to the current abstract state, and stops the analysis when the threshold is exceed.

1. The domain $D_{\mathbb{AE}}$ is an integer to determine the number of assume edges on the path: $D_{\mathbb{AE}} = (C, \mathcal{E}_{\mathbb{AE}}, [\![\cdot]\!])$, with $\mathcal{E}_{\mathbb{AE}} = ((\mathbb{N} \times \mathbb{B}), \sqsubseteq)$, $(e, t) \sqsubseteq (e', t')$ if $t = t'$ and $e = e'$.

   An abstract state consists of (1) an integer that stores how many abstract states have been computed for an assume edge on the path to the current state, and (2) a flag that shows whether number of abstract states computed for an assume edge on the path is larger than the user-specified threshold $k_{\mathbb{AE}}$.

2. The set $\Pi_{\mathbb{AE}}$ of precisions contains only one precision $\pi_{\mathbb{AE}}$.

3. The transfer relation $\rightsquigarrow_{\mathbb{AE}}$ has the transfer $(e, t) \overset{g}{\rightsquigarrow}_{\mathbb{AE}} ((e', t), \pi)$ for $g = (l, \cdot, l_s)$ if (1) $t' = (e' > k_{\mathbb{AE}})$ and (2) the following holds:

$$e' = \begin{cases} e + 1 & \text{if } g = (l, assume(p), l_s) \\ e & \text{else} \end{cases}$$

4. The merge operator combines elements by taking the maximum of the number of assume edges on the path when control flow meets: $\mathsf{merge}_{\mathbb{AE}}((e, t), (e', t'), \pi) = (e'', t'')$ such that we have $e'' = \max(e, e')$ and $t'' = t \vee t''$.

5. The termination check always returns *true* because the decision whether an abstract state is covered does not depend on the state of the conditions CPA.

6. The precision is never changed: $\mathsf{prec}_{\mathbb{AE}}(e, \pi, R) = (e, \pi)$.

We also implemented and used CPAs that force a time or memory limit on programs. These CPAs would ensure that the analysis terminates before failing and the results would be reported gracefully. Rather than working on individual abstract states, conditions that monitor the global progress force the analysis to terminate. We explain how we handle the imprecision inflicted by running the analysis with verification conditions in the next section.

## 5.4 Combining Verification Conditions and Assumptions

Verification conditions would work to limit and guide the exploration of state space of the program if used alone. But, in that case, they would produce imprecise results and the user would not have a detailed result of the analysis. For example, if the *CPA for repeating locations monitoring* was used, the analysis would not go deep in loops and it might have

stopped the analysis before exploring possible paths through some loops. The user would know that some loops might have been unfolded only a number of times which was specified by the threshold, but it would be a stronger verification report to include what abstract states have been produced under what conditions. That functionality is implemented with the *assumption collecting algorithm* and the *assumptions storage CPA*.

Since conditions are formalized and implemented as CPAs, they can be easily composed with other CPAs and the *assumptions storage CPA*. We can use the *assumption collecting algorithm* presented in Algorithm 4.

### 5.4.1 Composite CPA for Conditions and Assumptions

The composite CPA combines an existing CPA like the predicate analysis CPA $\mathbb{P}$ or the CPA $\mathbb{E}$ for explicit analysis with the assumption storage CPA $\mathbb{A}$, the CPA for location monitoring $\mathbb{L}$, and one CPA for conditions that limits the search space. The definition of the composite CPA can easily be extended in order to have several condition CPAs run in parallel by adding more component CPAs.

We present here, as an example the composite CPA for predicate analysis (using SBE for simplicity) together with the CPA for monitoring of repeating locations. Other combinations are similar. In fact, only a single operator (the strengthen operator $\downarrow$) and the merge strategy need to be changed in order to adopt the CPA to other configurations.

1. The domain $D_{\mathbb{C}}$ is defined by the cross product of the domains of $\mathbb{A}$, $\mathbb{L}$, $\mathbb{R}$ and $\mathbb{P}$.

2. The set of precisions is the cross product of the precision sets of the component CPAs: $\Pi_{\mathbb{C}} = \Pi_{\mathbb{A}} \times \Pi_{\mathbb{L}} \times \Pi_{\mathbb{R}} \times \Pi_{\mathbb{P}}$

3. The transfer relation $\rightsquigarrow_{\mathbb{C}}$ is the cross product of the transfer relations of the composite CPAs, but with the strengthen operator $\downarrow$ applied on the successor states:

$$\begin{aligned}\rightsquigarrow_{\mathbb{C}} = \{((e_{\mathbb{A}}, e_{\mathbb{L}}, e_{\mathbb{R}}, e_{\mathbb{P}}), g, (e''_{\mathbb{A}}, e''_{\mathbb{L}}, e''_{\mathbb{R}}, e''_{\mathbb{P}}), (\pi_{\mathbb{A}}, \pi_{\mathbb{L}}, \pi_{\mathbb{R}}, \pi_{\mathbb{P}})) \mid \\ (e_{\mathbb{A}}, g, e'_{\mathbb{A}}, \pi_{\mathbb{A}}) \in \rightsquigarrow_{\mathbb{A}}, (e_{\mathbb{L}}, g, e'_{\mathbb{L}}, \pi_{\mathbb{L}}) \in \rightsquigarrow_{\mathbb{L}}, \\ (e_{\mathbb{R}}, g, e'_{\mathbb{R}}, \pi_{\mathbb{R}}) \in \rightsquigarrow_{\mathbb{R}}, (e_{\mathbb{P}}, g, e'_{\mathbb{P}}, \pi_{\mathbb{P}}) \in \rightsquigarrow_{\mathbb{P}}, \\ (e''_{\mathbb{A}}, e_{\mathbb{L}}, e''_{\mathbb{R}}, e''_{\mathbb{P}}) = \downarrow(e'_{\mathbb{A}}, e'_{\mathbb{L}}, e'_{\mathbb{R}}, e'_{\mathbb{P}})\}\end{aligned}$$

   In the case the conditions CPA detects that a threshold is exceeded (for repeating locations monitoring this means that $e'_{\mathbb{R}} = (\cdot, \mathit{true})$), the strengthen operator adds an

assumption to the abstract state that excludes the current path: $\downarrow(e'_{\mathbb{A}}, e'_{\mathbb{L}}, e'_{\mathbb{R}}, e'_{\mathbb{P}}) = (\mathit{false}, e'_{\mathbb{L}}, e'_{\mathbb{R}}, e'_{\mathbb{P}})$ if $e'_{\mathbb{R}} = (\cdot, \mathit{true})$. Otherwise it is the identity function. If another CPA that generates assumptions (like the *overflow monitoring CPA*) is present, the strengthen operator would also add the assumptions generated by this CPA to the assumptions stored by the *assumption storage CPA*, and to the abstract state of the *predicate analysis CPA*.

4. The merge operator $\mathsf{merge}_{\mathbb{C}}$ merges elements, if the location and the predicate analysis part of them is equal:

$$\begin{aligned}&\mathsf{merge}((e_{\mathbb{A}}, e_{\mathbb{L}}, e_{\mathbb{R}}, e_{\mathbb{P}}), (e'_{\mathbb{A}}, e_{\mathbb{L}}, e'_{\mathbb{R}}, e_{\mathbb{P}}), (\pi_{\mathbb{A}}, \pi_{\mathbb{L}}, \pi_{\mathbb{R}}, \pi_{\mathbb{P}}))\\ &\quad = (\mathsf{merge}_{\mathbb{A}}(e_{\mathbb{A}}, e'_{\mathbb{A}}, \pi_{\mathbb{A}}), e_{\mathbb{L}}, \mathsf{merge}_{\mathbb{R}}(e_{\mathbb{R}}, e'_{\mathbb{R}}, \pi_{\mathbb{R}}), e_{\mathbb{P}})\end{aligned}$$

   Otherwise, it returns its second argument (i.e., does not merge).

   This relies on the fact that $\mathsf{merge}_{\mathbb{P}}$ never merges abstract states. If another CPA with a different merge implementation is used instead of $\mathbb{P}$, this needs to be adjusted. For example, if a CPA which always merges two abstract elements is used, $\mathsf{merge}_{\mathbb{C}}$ would always merge abstract elements with the same location.

5. The stop operator is the conjunction of the stop operators of the component CPAs: $\mathsf{stop}((e_{\mathbb{A}}, e_{\mathbb{L}}, e_{\mathbb{R}}, e_{\mathbb{P}}), R, (\pi_{\mathbb{A}}, \pi_{\mathbb{L}}, \pi_{\mathbb{R}}, \pi_{\mathbb{P}}))$
   $= \mathsf{stop}_{\mathbb{A}}(e_{\mathbb{A}}, R_{\mathbb{A}}, \pi_{\mathbb{A}}) \wedge \mathsf{stop}_{\mathbb{L}}(e_{\mathbb{L}}, R_{\mathbb{L}}, \pi_{\mathbb{L}}) \wedge \mathsf{stop}_{\mathbb{R}}(e_{\mathbb{R}}, R_{\mathbb{R}}, \pi_{\mathbb{R}}) \wedge \mathsf{stop}_{\mathbb{P}}(e_{\mathbb{P}}, R_{\mathbb{P}}, \pi_{\mathbb{P}})$

   where $R_{\mathbb{A}}$, $R_{\mathbb{L}}$, $R_{\mathbb{R}}$ and $R_{\mathbb{P}}$ are the projections of $R$ to the respective parts of the tuple that form an abstract state.

6. The precision adjustment operator uses the respective operator of the component CPAs:

$$\mathsf{prec}((e_{\mathbb{A}}, e_{\mathbb{L}}, e_{\mathbb{R}}, e_{\mathbb{P}}), (\pi_{\mathbb{A}}, \pi_{\mathbb{L}}, \pi_{\mathbb{R}}, \pi_{\mathbb{P}}), R) = ((e'_{\mathbb{A}}, e'_{\mathbb{L}}, e'_{\mathbb{R}}, e'_{\mathbb{P}}), (\pi'_{\mathbb{A}}, \pi'_{\mathbb{L}}, \pi'_{\mathbb{R}}, \pi'_{\mathbb{P}}))$$

$$\begin{aligned}(e'_{\mathbb{A}}, \pi'_{\mathbb{A}}) &= \mathsf{prec}_{\mathbb{A}}(e_{\mathbb{A}}, \pi_{\mathbb{A}}, R_{\mathbb{A}}),\\ (e'_{\mathbb{L}}, \pi'_{\mathbb{L}}) &= \mathsf{prec}_{\mathbb{L}}(e_{\mathbb{L}}, \pi_{\mathbb{L}}, R_{\mathbb{L}}),\\ (e'_{\mathbb{R}}, \pi'_{\mathbb{R}}) &= \mathsf{prec}_{\mathbb{R}}(e_{\mathbb{R}}, \pi_{\mathbb{R}}, R_{\mathbb{R}}),\\ (e'_{\mathbb{P}}, \pi'_{\mathbb{P}}) &= \mathsf{prec}_{\mathbb{P}}(e_{\mathbb{P}}, \pi_{\mathbb{P}}, R_{\mathbb{P}}).\end{aligned}$$

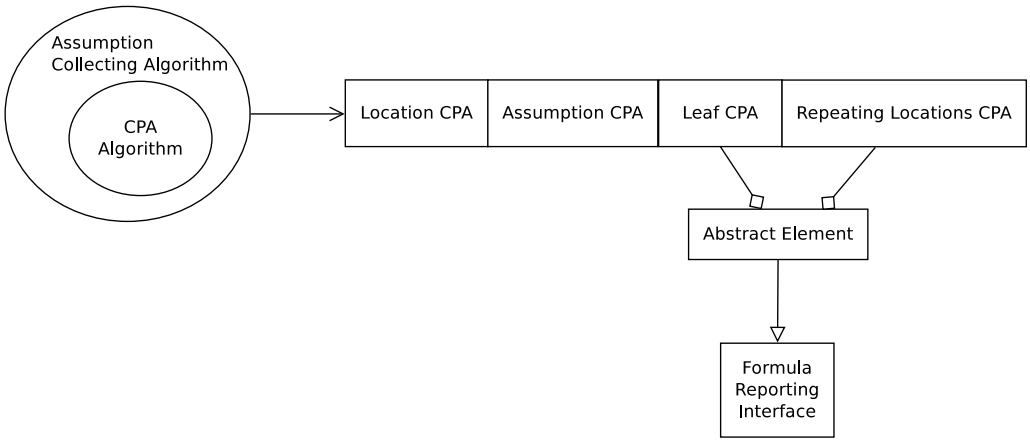


Figure 5.10: Architecture overview of assumption collecting algorithm using repeating locations monitoring CPA

In Figure 5.10, we show the overall architecture of the composition of several CPAs using assumptions and *repeating locations monitoring CPA*. Note that, any condition CPA can be added to the composite CPA. The strengthen operator is an integral part of the composite CPA when we use assumptions and conditions together. The strengthen operator, $\downarrow$ is used as a part of the transfer relation to compute a more precise abstract state [24, 81, 103]. For two abstract sets $E_1$ and $E_2$, $\downarrow : E_1 \times E_2 \rightarrow E_1$ computes a stronger element from the set $E_1$ using the element from $E_2$. Strengthening has to satisfy the soundness requirement $\downarrow(e, e') \sqsubseteq e$ [24]. Using strengthening in a composite CPA, an abstract element for a CPA can be adjusted to be more precise using another CPA's element rather than computing a pure cross product of elements. We use strengthening to adjust the abstract element for a condition CPA. If the threshold for a condition CPA is exceed, the *false* assumption is added for that state by $\downarrow$.

If we revisit the example explained in Figure 5.7, the process for the assumption generation can be followed. If the threshold for the *repeating locations monitoring CPA* is set to 9, after the node with location id 5 is visited for the $10^{th}$ time, the assumption element is adjusted to *false*. Thus, the abstract element for this location would be $e : (l =$

$5, \mathit{false}, (x = 10 \ \wedge \ y = 1), e_{\mathbb{R}})$ with location id 5, assumption *false*, the repeating locations element $e_{\mathbb{R}}$ and the explicit state $(x = 10 \ \wedge \ y = 1)$. Since the assumption is set to *false*, the analysis will not proceed due to requirements for transfer relation of *assumption storage CPA*. When the analysis terminates, the assumption collecting algorithm (given in Algorithm 4 in Section 5.2.2) will visit the `else` branch of the loop that iterates over the elements of reached and run the line assumption := assumption $\wedge (l = l_i \Rightarrow (\neg e_{\mathbb{T}} \vee e_{\mathbb{A}}))$;. In our example; $l_i = 5$, $e_{\mathbb{A}} = \mathit{false}$ and $e_{\mathbb{T}} = (x = 10 \ \wedge \ y = 1)$. So, the assumption $((l = 5) \Rightarrow (\neg(x = 10 \wedge y = 1) \vee \mathit{false}))$ which is equivalent to $(l = 5) \Rightarrow \neg(x = 10 \wedge y = 1)$ will be added to generated assumptions.

As shown in Figure 5.10, the *repeating locations monitoring CPA* also implements an abstract element that generates a formula. This formula is a simple uninterpreted function that does not change the meaning of the reported formula and it serves merely as a tool to report why the analysis was stopped after an element is visited. We assigned a representation to each condition CPA that we used and a list of these representations are shown in Table 5.1. The columns *Representation* lists how a condition is shown in a generated assumption and the columns *Example* and *Interpretation* show an example and describe what it means. For example for path length condition, $PL(100)$ states that the exploration of this path is stopped because the abstract states with this assumption was the $100^{th}$ abstract state on the path. If we revisit our example in Figure 5.7 again, the generated assumption was $(l = 5) \Rightarrow \neg(x = 10 \ \wedge \ y = 1)$. Since the analysis on this path was stopped due to repeating locations condition with the threshold value 9, we would add $RL(9)$ to the reported assumption. Thus the generated assumption will be $((l = 5) \Rightarrow \neg(x = 10 \wedge y = 1)) \wedge RL(9)$.

In Table 5.2, we show one example of conditions listed in Table 5.1. Explicit analysis times out for `diskperf_simpl1` from `ntdrivers` benchmarks as shown in Table 3.1. So, we used conditions with some benchmarks on the program and show the output for the given condition and threshold. The column *Conditions Used*, and *Threshold* list the name of the verification condition used and its threshold, respectively. $|R|$ gives the number of states in the set reached when the analysis terminates. $|L_{\mathbb{A}}|$ gives the number of locations with an assumption which is not *true*. For the number of locations specified by $|R|$, an assumption is generated and the analysis terminated by reporting an imprecise result. In the *Resulting Assumption* column, we show the assumption generated and reported at the end of the analysis. We show the locations that the condition has been generated for but we do not show the entire state for the explicit analysis and we represent it with a shorthand, $c_x$. Note

| Name | Representation | Example | Interpretation |
|---|---|---|---|
| Total Time | $TT()$ | $TT(30000)$ | *30000 ms = 30 s time-limit is hit* |
| Total Space | $TM()$ | $TM(40000000)$ | *400000000 bytes = 40 MB memory-limit is hit* |
| # Abstract States | $RE()$ | $RE(5000)$ | *Total # of abstract states exceeds 5000* |
| Path Length | $PL()$ | $PL(100)$ | *# of states on the path exceeds 100* |
| Repeating Locs. in Path | $RL()$ | $RL(10)$ | *A location on this path was seen 10 times* |
| Assume Edges in Path | $AE()$ | $AE(20)$ | *# of assume edges on this path exceeds 20* |

Table 5.1: Representations of conditions in an assumption formula

that, for the *Path Length* row, which shows the results for path length on path monitoring with a threshold of 500, we skipped the dumped formula due to its large size.

A human-readable report of one of the generated assumptions is presented in Figure 5.11. This is the assumption formula shown as $c_{24}$ in Table 5.2 for assume edges on path monitoring with threshold value 20. At the beginning of the file, we show the location before the implication sign `==>`. This assumption is for the location with id 650. There are two parts in the formula which are separated by conjunction `&`. That suggests that there were two different assumptions generated for this location and they were merged at the end. `AE(20)` at the end of both parts indicates that these assumptions were generated due to assume edges in path condition with threshold 20 (as shown in Table 5.1). Actually most of these two states that constitute two parts of the assumption share the same assignments for variables except the value assigned to `myStatus`. So, if this assumption is to be used later, it can be processed and simplified but to simply report it, we do not make expensive calls to decision solvers.

We can see common properties of reported assumptions for different conditions in Table 5.2. When we use the conditions for monitoring global progress (the first three rows), we

| Condition Used | Threshold | $\|R\|$ | $\|L_{\mathbb{A}}\|$ | Resulting Assumption |
|---|---|---|---|---|
| Total Time | 5000 | 12131 | 7 | $(l = 518 \Rightarrow c_1) \wedge (l = 489 \Rightarrow c_2) \wedge$ $(l = 296 \Rightarrow c_3) \wedge (l = 500 \Rightarrow c_4) \wedge$ $(l = 297 \Rightarrow c_5) \wedge (l = 287 \Rightarrow c_6) \wedge$ $(l = 268 \Rightarrow c_7)$ |
| Total Space | 400000000 | 35223 | 6 | $(l = 518 \Rightarrow c_8) \wedge (l = 489 \Rightarrow c_9) \wedge$ $(l = 500 \Rightarrow c_{10}) \wedge (l = 287 \Rightarrow c_{11}) \wedge$ $(l = 286 \Rightarrow c_{12}) \wedge (l = 268 \Rightarrow c_{13})$ |
| # Abstract States | 5000 | 5000 | 6 | $(l = 518 \Rightarrow c_{14}) \wedge (l = 489 \Rightarrow c_{15}) \wedge$ $(l = 500 \Rightarrow c_{16}) \wedge (l = 286 \Rightarrow c_{17}) \wedge$ $(l = 268 \Rightarrow c_{18}) \wedge (l = 284 \Rightarrow c_{19})$ |
| Path Length | 500 | 13312 | 39 | Skipped |
| Repeating Locs | 2 | 4955 | 2 | $(l = 66 \Rightarrow c_{20}) \wedge (l = 284 \Rightarrow c_{21})$ |
| Assume Edges | 20 | 3180 | 13 | $(l = 712 \Rightarrow c_{22}) \wedge (l = 369 \Rightarrow c_{23}) \wedge$ $(l = 650 \Rightarrow c_{24}) \wedge (l = 370 \Rightarrow c_{25}) \wedge$ $(l = 587 \Rightarrow c_{26}) \wedge (l = 287 \Rightarrow c_{27}) \wedge$ $(l = 288 \Rightarrow c_{28}) \wedge (l = 730 \Rightarrow c_{29}) \wedge$ $(l = 561 \Rightarrow c_{30}) \wedge (l = 560 \Rightarrow c_{31}) \wedge$ $(l = 600 \Rightarrow c_{32}) \wedge (l = 727 \Rightarrow c_{33}) \wedge$ $(l = 606 \Rightarrow c_{34})$ |

Table 5.2: Examples for using conditions with explicit analysis on `diskperf_simpl1`

can see the same locations on the assumption formula even though the number of abstract state in reached might differ. This might be evaluated as follows: The analysis creates an abstract state for these locations and after inserting them in waitlist, it spends a significant time on some other part of the ARG. The reason of the behaviour might be a loop or a recursive call. The last three columns show results for conditions that limit the path size. Including the skipped assumption formula of path length condition, these formulas do not include assumptions for locations 518, or 489 for example. That suggests that the analysis skips the re-entrant part when the path is limited with a condition and continues with the abstract states that were inserted to waitlist. Using verification conditions might be useful for analyzing the behaviour of the analysis and understanding the structure of the code this way.

```
l = 650 ==>
((!((compRegistered = 1) & (setEventCalled = 1) & (customIrp = 1) &
(Executive = 0) & (lowerDriverReturn = 0) & (IPC = 7) & (DC = 2) &
(KernelMode = 0) & (routine = 0) & (NP = 1) & (compFptr = 0) & (s = 1) &
(pended = 0) & (myStatus = 0) & (UNLOADED = 0) & (MPR3 = 6) &
(MPR1 = 5) & (SKIP1 = 3) & (SKIP2 = 4)) & AE(20))
&
(!((compRegistered = 1) & (setEventCalled = 1) & (customIrp = 1) &
(Executive = 0) & (lowerDriverReturn = 0) & (IPC = 7) & (DC = 2) &
(KernelMode = 0) & (routine = 0) & (NP = 1) & (compFptr = 0) & (s = 1) &
(pended = 0) & (myStatus = -1073741637) & (UNLOADED = 0) & (MPR3 = 6) &
(MPR1 = 5) & (SKIP1 = 3) & (SKIP2 = 4)) & AE(20)))
```

Figure 5.11: An example assumption in human-readable format for the condition $c_{24}$ in Table 5.2

## 5.5 Verification Conditions for Bug Detection

We have used some conditions on some buggy programs to test their capabilities. Different threshold values for different conditions were used for experiments and we reported the ones that increased the performance significantly. Table 5.3 reports performance results for explicit analysis for programs with a known bug. The time limit was set to 15 min as in the previous tables. Some parts of the code that implements verification conditions are from revision 4257 of CPAchecker.

The column *Program* lists the name of the used program. The column *Condition & Threshold* lists the used condition and the corresponding threshold value. Conventional model checking fails to find the violation of the specification within the specified time limit for 4 programs and it takes 289 s for `test_locks_14.BUG` and 529 s for `transmitter.15.BUG`. Model checking using conditions with explicit analysis with the condition 'Path Length' or with the condition 'Repeating Locations in Path' identify the error in all examples. The experiments show a significant performance improvement: For instance `sfifo_1_BUG` and `sfifo_2_BUG` were falsified within a few seconds, and `transmitter` examples took only a few seconds.

**Discussion.** Using conditions demonstrates that especially if the verification task is run under limited resources, excluding some parts of the code to run a partial verification can be very useful. Monitoring the code until resources are exhausted helps us to run the verification

| Config. | Program | Conventional MC Result | | Conditional MC Condition & Threshold | Result | |
|---|---|---|---|---|---|---|
| Explicit Analysis | `pc_sfifo_1_BUG` | 900 | - | Repeating Loc. in Path: 3 | 1.77 | ✓ |
| | `pc_sfifo_2_BUG` | 900 | - | Repeating Loc. in Path: 3 | 1.86 | ✓ |
| | `test_locks_14.BUG` | 289 | ✓ | Path Length: 85 | 33.3 | ✓ |
| | `test_locks_15.BUG` | 543 | - | Path Length: 90 | 58.2 | ✓ |
| | `transmitter.15.BUG` | 529 | ✓ | Path Length: 600 | 5.19 | ✓ |
| | `transmitter.16.BUG` | 900 | - | Path Length: 600 | 2.77 | ✓ |

Table 5.3: Conventional versus conditional analysis on programs with a bug

using different conditions. Conditions for global progress is beneficial to terminate the analysis before the tool crashes and to use the results produced so far. Trying different conditions on programs that a traditional technique fails to analyze and evaluating the output gives an idea of the structure of the code and what to do with the analysis. In some cases, concentrating on a certain part of the program helps detecting a bug very fast. For example, a bug was found within a few seconds when an appropriate condition was used where the traditional methods failed to detect the bug within the given time limit.

We experimented with different conditions after a timeout to solve the verification problem. The most suitable threshold for conditions are determined based on our previous runs by evaluating the verification reports at the end of the analysis. In the next section, we present a framework that can be used to estimate a threshold value for the condition automatically. The automation for this process will help us to verify more programs under limited resources.

## 5.6 Adjustable Conditions

In this section, we present a framework that runs conditional model checking automatically by adjusting the thresholds. We first explain the motivation and then describe the algorithm that runs conditional software analysis with adjustable conditions. Then we run experiments on our benchmark programs.

### 5.6.1 Motivation

In Table 5.3, we have presented experiments for conditional model checking which shows that using verification conditions can significantly improve the run-time for analysis. We have

analyzed some programs with different threshold values for several conditions and presented the experiments that improved the results dramatically. This is not a practical approach in some cases. Sometimes even slight changes in the threshold value may effect the efficiency of the analysis. For example, for `test_locks_14.BUG`, we set the threshold for path length condition to 85, and for `test_locks_15.BUG`, we set it to 90. Using the same threshold did not lead to achieve to a successful analysis for these two programs. Additionally, it can be observed in Table 5.3 that a good choice of threshold value depends on the structure of the program. We use the same condition, path length, for `test_locks`* and `transmitter` examples, but threshold values that we used is 6 times larger for `transmitter`.

So, to have practical usability of verification conditions, especially for bug detection, we developed a technique where we run the analysis on a program and monitor the progress. We use our sequential composition framework to update or adjust the threshold values. We limit a run of the analysis with a condition. Then, we collect statistics about the progress of the analysis within the given limit. In the next step, the analysis evaluates the statistics collected and restarts the analysis by adjusting the conditions.

This framework is particularly useful for bug detection and for exploring bigger coverage of the state space using verification conditions. If the problem for the analysis is spending a significant time on a particular subset of the ARG and not exploring other parts exhaustively, that can be detected by checking the statistics from previous runs. If the updated threshold value is not large enough, then the analysis will adjust it again and restart the algorithm. This approach helps the developer to broaden the state space search not only based on a particular search strategy but based on the behaviour of the program on-the-fly. Another advantage of our approach is its compatibility since the conditions are provided as CPAs. They can be used with any technique and algorithm that is encoded as a CPA and the user is not required to change the implementation of the analysis CPA or interact with the process to use a condition with a suitable threshold value. It would be enough to specify which CPAs and which conditions to be used for the analysis by only providing a configuration file.

### 5.6.2 Algorithm and Implementation

We implemented adjustable conditions algorithm similar to sequential composition framework. It is actually an extended version of the sequential composition algorithm.

**Algorithm 5** *AdjustableConditions*$(A, \mathbb{C}, R_0, W_0)$

**Input:** an algorithm $A$ that uses the CPA $\mathbb{C}$
a CPA $\mathbb{C}$ including assumptions storage CPA and progress observer CPA ,
a set $R_0 \subseteq E$ of abstract states,
a subset $W_0 \subseteq R_0$ of frontier abstract states,
**Output:** a set of reachable abstract states,
a subset of frontier abstract states
**Variables:** two sets reached and waitlist, boolean adjusted to track adjustment status

```
reached := R_0;
waitlist := W_0;
adjusted := true
while adjusted do
    adjusted := false
    (reached, waitlist) := A.run(reached, waitlist);
    assumptionElementsList := getElementsWithAssumptions(reached, waitlist)
    if |assumptionElementsList| > 0 then
        for each e ∈ assumptionElementsList do
            reached := (reached) \ {e};
            parent := e.getParent();
            // adjustThreshold() adjusts the threshold for the abstract element
            // and if nothing is changed return false
            adjusted := adjusted ∨ parent.adjustThreshold();
            waitlist := waitlist ∪ {parent};
return (reached, waitlist)
```

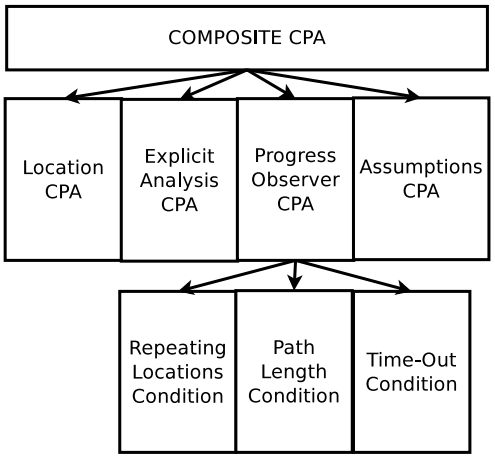


Figure 5.12: CPA with progress observer CPA including *Repeating Locations*, *Path Length* and *Time-out*, and assumptions storage CPA.

Algorithm 5 shows the procedure that implements verification with adjustable conditions. The algorithm gets as input, another CPA algorithm (usually a assumption collecting algorithm) which is referred as the inner algorithm. The inner algorithm runs on a composite CPA that includes a progress observer CPA and an assumptions CPA with the hierarchical structure shown in Figure 5.12.

Progress observer CPA can be regarded as a composite CPA which composes several conditions CPAs and observes the behavior of the analysis. It collect statistics about the analysis and guides the execution of the analysis using verification conditions which are specified by the user. Assumptions CPA collects and reports the assumptions during the analysis. For the adjustable conditions algorithm, we use assumptions CPA to determine if the analysis was stopped by the progress observer CPA.

reached and waitlist is the initial sets that are passed to the inner algorithm and updated each time the inner algorithm is run. The boolean variable `adjusted` decides to continue to run the analysis if the threshold of the progress observer element is updated. The 'while' loop continues until `adjusted` is set to *false*. The variable `adjusted` is set to *false* initially at the beginning of each iteration (at line 5) and it is updated at line 14. At line 6, the inner

algorithm is run using the given waitlist and reached, and these two sets are updated as the algorithm returns the updated sets. As explained before, when the progress observer CPA and the assumption storage CPA are used together, an assumption is created and saved for an abstract element which stops the analysis after the threshold for a condition is exceed.

At line 7, we put all elements with an assumption formula which is different than *true* to the list `assumptionElementsList`. At Line 8, we check if there are any elements in `assumptionElementsList`. If `assumptionElementsList` is empty, it means that there were no elements with an assumption formula different than *true* and since the variable `adjusted` is not updated, the 'while' loop terminates and the algorithm reports the result of the CPA algorithm that is run at line 6. Between lines 9 and 15, we adjust the thresholds of conditions. First, if there are some elements in `assumptionElementsList`, we traverse them and remove them from the reached set so that we will add them back to waitlist after updating their thresholds and continue analysis. Then, we call the threshold adjustment function for condition thresholds for parents (parent of the abstract element on the ARG) of these elements with assumptions. For example, if the threshold for CPA for Repeating Locations Monitoring was set to 3, we want to increase the threshold by some predefined amount (let us assume that we want to double the threshold value and make it 6) and if this operation adjusts the threshold successfully, we return true. So, if any parents' threshold is adjusted, the variable `adjusted` becomes true and the 'while' loop continues with the next iteration. At the last line of the 'for' loop, we add the adjusted parent back to waitlist. This ensures that the inner algorithm at line 6 will continue analyzing the program using re-added parents with their new threshold values for given conditions. The algorithm ends when no thresholds are adjusted for any elements, so the 'while loop' terminates and returns reached and waitlist.

The function $adjustThreshold()$ is called for each `parent` at line 14, and it is handled by the progress observer CPA shown in Figure 5.12. The CPA delegates this call to each condition provided to the CPA. For example in Figure 5.12, three conditions are included, namely *repeating locations monitoring CPA*, *path length monitoring CPA* and *time-out CPA*. Each of these conditions provides their own methods of threshold adjustment functions and they update the threshold accordingly. If any of the included conditions' threshold adjustment function adjusts its threshold and returns true, $adjustThreshold()$ of progress observer CPA returns true. If all of them return false, this means that none of the conditions adjusted its threshold for the given progress observer element.

### 5.6.3 Threshold Adjustment for Conditions

We have used three different conditions with our experiments and provided implementations for condition adjusting methods for them. Experiments have shown that for our benchmark programs, using a limit of 10 s produces the best results. So, we will introduce threshold adjustment functions that are used with the 10 s time-limit condition with explicit analysis CPA. It should be noted that these adjustment steps are configurable and can be modified and used with any CPA or any condition based on the user needs.

We have run experiments using explicit analysis with the 10 s time limit and compared performances of different threshold values for different conditions and provided the adjustment functions and configured them based on these experiments:

- **Threshold Adjustment for Path Length Monitoring CPA**: We start the analysis using a CPA that includes explicit analysis with threshold value $k_{PL} = \infty$ and 10 s time limit. After the analysis terminates for this configuration, we extract the maximum size of a path of the ARG, and assign this value to the variable maxPL. The initial threshold, initPL is set to maxPL/5. After the initial threshold initPL is set, the algorithm restarts the analysis. Following its termination, the threshold is adjusted again if needed (based on the condition at line 8 of Algorithm 5). After the initialization step, the threshold is increased by a value determined by the variable incPL = initPL/4. So, the increase rate is also based on the variable maxPL.

- **Threshold Adjustment for Repeating Locations Monitoring CPA**: We use a similar strategy as the path length monitoring condition. Starting with $k_{RL} = \infty$, the maximum number of repetitions of a variable on a path is saved as maxRL. Then the initial threshold values is determined using this maximum value, initRL = maxRL/5. After the initialization, at each step of the adjustment, the threshold is increased by initRL, incRL = initRL.

- **Threshold Adjustment for Assume Edges on Path Monitoring CPA**: This condition CPA also takes a similar approach to adjust its threshold. After the analysis terminates with the time limit, we get the maximum number of assume edges on a path of the ARG, and assign this value to the variable maxAE. The initial threshold, initAE is set to maxAE/5. After the initialization step, the threshold is increased by a value determined by the variable incAE = initAE/4.

In the next section, we present experiments using this configuration with 10 s time limit condition.

## 5.7 Experiments for Adjustable Conditions

We experimented with explicit analysis using several configurations. Since explicit analysis may report false alarms at some cases, we use CBMC to verify the feasibility of the error path in all configurations. So, when a bug is reported, the error path reported is also analyzed by CBMC and only if CBMC confirms that it is indeed an error path, the bug is reported. For the experiments, we used the default explicit analysis configuration in CPAchecker with no conditions and then we run it by using three different adjustable conditions.

Configurations with conditions adjust their threshold values using a time-out condition. We limited a single run of the program by 10 s using a time-limit condition. In the first run, the threshold for the used condition is not set. So, the program runs only using the 10 s time-out condition and collects statistics to determine the initial threshold for the given condition. Then, if the analysis does not return an answer within the 10 s time-limit, the analysis stops and sets the initial threshold value for the enabled condition. The analysis is then restarted with the selected condition and initial threshold value. Note that the time-out condition with 10 s limit is still used and there are three possible scenarios after that step:

1. The analysis might terminate and return and answer.

2. The analysis might hit the 10 s time limit and adjust the threshold of the enabled conditions. Each condition provides its own method of adjusting the threshold which is explained in the previous section.

3. The analysis might stop before 10 s time limit because the threshold might be too small and it might have stopped after exceeding the threshold value for enabled thresholds. In that case, the analysis adjusts the thresholds and restart the analysis.

### 5.7.1 Adjustable Conditions on Benchmark Programs

In Table 5.4, we present results using explicit analysis and several conditions. The tables show the run-times consumed by the CPU in seconds. We use a 3 min global time limit for

Table 5.4: Adjustable conditions

| | No Conditions | | Path Length | | | Rept. in Path | | | Assume Edges | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Res. | *Time* | Res. | *Time* | Thr. | Res. | *Time* | Thr. | Res. | *Time* | Thr. |
| `test_locks_14.BUG` | - | 180 | ✓ | 11.3 | 29 | - | 180 | - | ✓ | 11.3 | 8 |
| `test_locks_15.BUG` | - | 180 | ✓ | 154 | 31 | - | 180 | - | ✓ | 11.3 | 9 |
| `test_locks_5` | ✓ | 1.52 | ✓ | 1.63 | - | ✓ | 1.61 | - | ✓ | 1.62 | - |
| `test_locks_6` | ✓ | 2.08 | ✓ | 2.38 | - | ✓ | 1.95 | - | ✓ | 2.37 | - |
| `test_locks_7` | ✓ | 1.92 | ✓ | 2.22 | - | ✓ | 2.24 | - | ✓ | 2.20 | - |
| `test_locks_8` | ✓ | 2.82 | ✓ | 2.82 | - | ✓ | 2.82 | - | ✓ | 2.83 | - |
| `test_locks_9` | ✓ | 3.35 | ✓ | 4.50 | - | ✓ | 4.05 | - | ✓ | 4.16 | - |
| `test_locks_10` | ✓ | 6.28 | ✓ | 8.09 | - | ✓ | 7.81 | - | ✓ | 8.27 | - |
| `test_locks_11` | ✓ | 25.0 | ✓ | 133 | 84 | ✓ | 29.4 | - | - | 180 | - |
| `test_locks_12` | ✓ | 122 | - | 180 | - | ✓ | 148 | - | - | 180 | - |
| `test_locks_13` | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| `test_locks_14` | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| `test_locks_15` | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| **Locks total** | 8 | 1060 | 9 | 1040 | - | 8 | 1100 | - | 8 | 944 | - |
| `cdaudio_simpl1_BUG` | ✓ | 2.99 | ✓ | 2.75 | - | ✓ | 3.34 | - | ✓ | 2.89 | - |
| `floppy_simpl3_BUG` | ✓ | 2.47 | ✓ | 2.26 | - | ✓ | 2.20 | - | ✓ | 2.25 | - |
| `floppy_simpl4_BUG` | ✓ | 2.58 | ✓ | 2.67 | - | ✓ | 2.66 | - | ✓ | 2.52 | - |
| `kbfiltr_simpl2_BUG` | ✓ | 2.15 | ✓ | 2.64 | - | ✓ | 2.20 | - | ✓ | 2.18 | - |
| `cdaudio_simpl1` | ✓ | 3.67 | ✓ | 4.40 | - | ✓ | 3.57 | - | ✓ | 3.99 | - |
| `diskperf_simpl1` | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| `floppy_simpl3` | ✓ | 2.62 | ✓ | 2.91 | - | ✓ | 2.96 | - | ✓ | 2.91 | - |
| `floppy_simpl4` | ✓ | 3.32 | ✓ | 4.15 | - | ✓ | 3.80 | - | ✓ | 4.12 | - |
| `kbfiltr_simpl1` | ✓ | 2.39 | ✓ | 3.16 | - | ✓ | 2.60 | - | ✓ | 2.64 | - |
| `kbfiltr_simpl2` | ✓ | 2.89 | ✓ | 3.71 | - | ✓ | 3.19 | - | ✓ | 3.14 | - |
| **NT drivers total** | 9 | 205 | 9 | 209 | - | 9 | 207 | - | 9 | 207 | - |
| `s3_clnt_1_BUG` | ✓ | 4.17 | ✓ | 4.29 | - | ✓ | 4.37 | - | ✓ | 4.90 | - |
| `s3_clnt_2_BUG` | ✓ | 4.15 | ✓ | 4.69 | - | ✓ | 4.38 | - | ✓ | 4.31 | - |
| `s3_clnt_3_BUG` | ✓ | 3.99 | ✓ | 4.38 | - | ✓ | 5.03 | - | ✓ | 4.83 | - |
| `s3_clnt_4_BUG` | ✓ | 3.77 | ✓ | 4.31 | - | ✓ | 4.35 | - | ✓ | 4.85 | - |
| `s3_srvr_11_BUG` | ✓ | 2.57 | ✓ | 2.28 | - | ✓ | 2.29 | - | ✓ | 2.29 | - |
| `s3_srvr_12_BUG` | ✓ | 2.24 | ✓ | 2.59 | - | ✓ | 2.62 | - | ✓ | 2.24 | - |
| `s3_srvr_14_BUG` | ✓ | 2.60 | ✓ | 2.65 | - | ✓ | 2.79 | - | ✓ | 2.65 | - |
| `s3_srvr_1_BUG` | ✓ | 1.83 | ✓ | 1.88 | - | ✓ | 2.18 | - | ✓ | 1.84 | - |
| `s3_srvr_2_BUG` | ✓ | 2.09 | ✓ | 2.16 | - | ✓ | 1.80 | - | ✓ | 1.79 | - |
| `s3_srvr_6_BUG` | - | 180 | ✓ | 18.9 | 729 | ✓ | 20.3 | 9 | ✓ | 113 | 232 |
| `s3_clnt_1` | ✓ | 8.07 | ✓ | 10.5 | - | ✓ | 9.71 | - | ✓ | 9.68 | - |
| `s3_clnt_2` | ✓ | 7.86 | ✓ | 9.80 | - | ✓ | 9.73 | - | ✓ | 10.4 | - |
| `s3_clnt_3` | ✓ | 8.99 | ✓ | 9.80 | - | ✓ | 9.69 | - | ✓ | 9.86 | - |
| `s3_clnt_4` | ✓ | 8.29 | ✓ | 9.78 | - | ✓ | 9.76 | - | ✓ | 9.96 | - |
| `s3_srvr_1` | ✓ | 2.58 | ✓ | 2.88 | - | ✓ | 2.89 | - | ✓ | 2.90 | - |
| `s3_srvr_1a` | - | 2.91 | - | 2.62 | - | - | 2.68 | - | - | 2.65 | - |

Continued on next page

**Table 5.4 – continued from previous page**

| | No Conditions | | Path Length | | | Rept. in Path | | | Assume Edges | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Res. | *Time* | Res. | *Time* | Thr. | Res. | *Time* | Thr. | Res. | *Time* | Thr. |
| s3_srvr_1b | - | 1.84 | - | 2.14 | - | - | 1.86 | - | - | 1.86 | - |
| s3_srvr_2 | ✓ | 2.87 | ✓ | 2.76 | - | ✓ | 2.76 | - | ✓ | 3.37 | - |
| s3_srvr_3 | - | 2.73 | - | 3.04 | - | - | 3.61 | - | - | 3.05 | - |
| s3_srvr_4 | - | 2.70 | - | 3.16 | - | - | 3.69 | - | - | 3.06 | - |
| s3_srvr_6 | - | 180 | - | 180 | - | - | 180 | - | ✓ | 136 | 232 |
| s3_srvr_7 | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| s3_srvr_8 | ✓ | 2.52 | ✓ | 3.21 | - | ✓ | 2.87 | - | ✓ | 3.47 | - |
| **SSH total** | 16 | 619 | 17 | 468 | - | 17 | 470 | - | 18 | 519 | - |
| kundu1_BUG | ✓ | 2.24 | ✓ | 2.04 | - | ✓ | 2.06 | - | ✓ | 2.04 | - |
| kundu2_BUG | ✓ | 1.89 | ✓ | 1.88 | - | ✓ | 1.64 | - | ✓ | 1.64 | - |
| pc_sfifo_1_BUG | - | 180 | ✓ | 16.0 | 12641 | ✓ | 30.7 | 184 | ✓ | 30.9 | 2419 |
| pc_sfifo_2_BUG | - | 180 | ✓ | 19.2 | 18114 | ✓ | 46.0 | 233 | ✓ | 75.6 | 5143 |
| pipeline_BUG | - | 11.0 | - | 13.5 | - | - | 13.9 | - | - | 13.0 | - |
| token_ring.01.BUG | - | 1.95 | - | 1.85 | - | - | 1.82 | - | - | 2.06 | - |
| token_ring.02.BUG | - | 2.06 | - | 2.41 | - | - | 2.36 | - | - | 2.03 | - |
| token_ring.03.BUG | - | 2.60 | - | 2.58 | - | - | 2.62 | - | - | 2.59 | - |
| token_ring.04.BUG | - | 2.90 | - | 3.28 | - | - | 3.60 | - | - | 3.21 | - |
| token_ring.05.BUG | - | 3.81 | - | 4.42 | - | - | 4.83 | - | - | 3.96 | - |
| token_ring.06.BUG | - | 4.94 | - | 5.58 | - | - | 5.55 | - | - | 6.19 | - |
| token_ring.07.BUG | - | 8.83 | - | 9.68 | - | - | 10.5 | - | - | 10.3 | - |
| token_ring.08.BUG | - | 29.1 | - | 26.2 | 415 | - | 30.1 | 2 | - | 28.8 | 81 |
| token_ring.09.BUG | - | 125 | - | 143 | 486 | - | 147 | 2 | - | 147 | 96 |
| token_ring.10.BUG | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| token_ring.11.BUG | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| token_ring.12.BUG | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| token_ring.13.BUG | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| token_ring.14.BUG | - | 180 | ✓ | 35.6 | 839 | - | 180 | - | ✓ | 37.1 | 224 |
| token_ring.15.BUG | - | 180 | ✓ | 36.7 | 1230 | - | 180 | - | ✓ | 39.7 | 266 |
| toy1_BUG | - | 2.78 | - | 2.73 | - | - | 2.75 | - | - | 2.78 | - |
| toy2_BUG | ✓ | 2.21 | ✓ | 2.30 | - | ✓ | 2.22 | - | ✓ | 2.22 | - |
| transmitter.01.BUG | ✓ | 1.55 | ✓ | 1.66 | - | ✓ | 1.64 | - | ✓ | 1.66 | - |
| transmitter.02.BUG | ✓ | 1.73 | ✓ | 1.82 | - | ✓ | 1.83 | - | ✓ | 1.87 | - |
| transmitter.03.BUG | ✓ | 1.97 | ✓ | 2.04 | - | ✓ | 2.03 | - | ✓ | 2.41 | - |
| transmitter.04.BUG | ✓ | 2.26 | ✓ | 2.64 | - | ✓ | 2.27 | - | ✓ | 2.62 | - |
| transmitter.05.BUG | ✓ | 2.46 | ✓ | 2.49 | - | ✓ | 2.50 | - | ✓ | 2.50 | - |
| transmitter.06.BUG | ✓ | 2.79 | ✓ | 3.61 | - | ✓ | 3.10 | - | ✓ | 3.11 | - |
| transmitter.07.BUG | ✓ | 3.65 | ✓ | 3.87 | - | ✓ | 3.80 | - | ✓ | 3.73 | - |
| transmitter.08.BUG | ✓ | 5.23 | ✓ | 6.41 | - | ✓ | 5.88 | - | ✓ | 5.81 | - |
| transmitter.09.BUG | ✓ | 10.3 | ✓ | 10.7 | - | ✓ | 10.8 | - | ✓ | 10.7 | - |
| transmitter.10.BUG | ✓ | 28.0 | ✓ | 29.8 | 828 | ✓ | 34.0 | - | ✓ | 31.9 | 198 |
| transmitter.11.BUG | ✓ | 124 | ✓ | 33.7 | 1038 | ✓ | 150 | - | ✓ | 33.3 | 221 |
| transmitter.12.BUG | - | 180 | ✓ | 36.5 | 1200 | - | 180 | - | ✓ | 36.6 | 200 |

Continued on next page

**Table 5.4 – continued from previous page**

| | No Conditions | | Path Length | | | Rept. in Path | | | Assume Edges | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Res. | *Time* | Res. | *Time* | Thr. | Res. | *Time* | Thr. | Res. | *Time* | Thr. |
| `transmitter.13.BUG` | - | 180 | ✓ | 36.7 | 1106 | - | 180 | - | ✓ | 38.2 | 260 |
| `transmitter.15.BUG` | - | 180 | ✓ | 16.4 | 1106 | - | 180 | - | ✓ | 17.2 | 80 |
| `transmitter.16.BUG` | - | 180 | ✓ | 16.7 | 1443 | - | 180 | - | ✓ | 16.7 | 85 |
| `bist_cell` | ✓ | 1.47 | ✓ | 1.47 | - | ✓ | 1.46 | - | ✓ | 1.47 | - |
| `kundu` | ✓ | 6.05 | ✓ | 7.37 | - | ✓ | 7.52 | - | ✓ | 7.83 | - |
| `mem_slave_tlm.1` | - | 2.17 | - | 2.55 | - | - | 2.13 | - | - | 2.14 | - |
| `mem_slave_tlm.2` | - | 2.46 | - | 2.50 | - | - | 2.48 | - | - | 2.49 | - |
| `mem_slave_tlm.3` | - | 2.73 | - | 2.74 | - | - | 2.78 | - | - | 2.80 | - |
| `mem_slave_tlm.4` | - | 3.18 | - | 3.40 | - | - | 3.23 | - | - | 3.29 | - |
| `mem_slave_tlm.5` | - | 3.58 | - | 3.92 | - | - | 4.22 | - | - | 4.43 | - |
| `pc_sfifo_1` | - | 180 | - | 180 | - | - | 180 | 2046 | - | 180 | 19267 |
| `pc_sfifo_2` | - | 180 | - | 180 | - | - | 180 | 2330 | - | 180 | 18392 |
| `pc_sfifo_3` | ✓ | 1.63 | ✓ | 1.67 | - | ✓ | 2.00 | - | ✓ | 1.67 | - |
| `pipeline` | - | 11.2 | - | 13.9 | - | - | 14.1 | - | - | 13.6 | - |
| `token_ring.01` | - | 1.61 | - | 1.67 | - | - | 1.65 | - | - | 1.67 | - |
| `token_ring.02` | - | 1.82 | - | 1.95 | - | - | 2.26 | - | - | 1.91 | - |
| `token_ring.03` | - | 2.17 | - | 2.72 | - | - | 2.29 | - | - | 2.28 | - |
| `token_ring.04` | - | 2.67 | - | 3.54 | - | - | 3.49 | - | - | 2.90 | - |
| `token_ring.05` | - | 3.66 | - | 4.48 | - | - | 3.95 | - | - | 4.49 | - |
| `token_ring.06` | - | 5.53 | - | 5.62 | - | - | 5.62 | - | - | 5.67 | - |
| `token_ring.07` | - | 8.84 | - | 9.28 | - | - | 10.3 | - | - | 9.73 | - |
| `token_ring.08` | - | 27.4 | - | 28.6 | 414 | - | 27.8 | 2 | - | 29.4 | 80 |
| `token_ring.09` | - | 96.3 | - | 132 | 486 | - | 128 | 2 | - | 144 | 95 |
| `token_ring.10` | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| `token_ring.11` | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| `token_ring.12` | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| `token_ring.13` | - | 180 | - | 180 | - | - | 180 | - | - | 180 | - |
| `toy` | - | 2.34 | - | 2.4 | - | - | 2.36 | - | - | 2.34 | - |
| **SystemC total** | 17 | 3810 | 25 | 2570 | - | 19 | 3630 | - | 25 | 2660 | - |
| **Total** | 50 | 5700 | 60 | 4280 | - | 53 | 5410 | - | 60 | 4330 | |

all configurations. The column 'Program' lists the benchmark programs used. Under 'No Condition', we report results with only explicit analysis (no verification conditions are used). The other three columns report results for 'Path Length Monitoring CPA', 'Repeating Locations in Path Monitoring CPA' and 'Assume Edges in Path Monitoring CPA'. The column 'Trh.' shown the threshold value for the corresponding condition when the analysis ends. For instance, when we used 'Path Length Condition' for analyzing `test_locks_14.BUG`, the threshold value for that condition was set to 29 and the analysis reported UNSAFE using this condition. A dash on 'Trh.' column indicates that either the threshold was never set or the analysis timed-out and did not report a result. The threshold might not be set for two reasons: (i) The analysis might take less than 10 s to terminate, so the initial threshold value is never set. (ii) The condition's threshold adjustment method did not set the initial value because it does not have required statistical information. For example, for 'Repeating Locations in Path Condition', we use the number of maximum occurrences of a CFA node on an ARG path to set the initial value for the threshold. If no loops are unfold during the analysis for a program, the maximum occurrence of a node on a path is 1 and the threshold is never initialized as in the case of `test_locks_14.BUG`.

At the bottom of the table, we show total number of successfully analyzed programs and the total run-time of the analysis for all programs. Also, after each benchmark set, we show totals for this benchmark set. As seen in Table 5.4, all three configurations that use adjustable conditions can analyze more programs successfully compared to using only explicit analysis with no conditions. Path length and assume edges conditions with adjustable threshold adjustment can analyze 10 more programs compared to analysis with no conditions. Furthermore, using adjustable conditions significantly improved the total run-time when we used path length condition or assume edges in path condition. For all 108 programs, the analysis used 5700 s when no conditions is used and using adjustable path length condition can run all these programs in 4280 s, as well as adjustable assume edges in path condition spends 4330 s in total.

For the benchmark set `locks`, only path length condition is able to analyze 9 programs whereas other configurations can analyze 8 programs. Among all programs that are analyzed, only the analyses on `test_locks_12` performed better than path length and assume edges conditions when no conditions is used. However, repeating locations in path can analyze this program although it spends more time. Path length condition and assume edges

condition can analyze `test_locks_14.BUG` and `test_locks_15.BUG` and path length condition can analyze `test_locks_11` additionally. Comparing total run-times and number of analyzed programs reveals that using adjustable conditions does not negatively effect the results of explicit analysis for `locks` benchmark programs.

Programs in the `ntdrivers` set presents an interesting case: As shown in the table, these programs can be verified using all configurations and using adjustable conditions and there is only one program which cannot be analyzed by any configurations which is `diskperf_simpl1`. Total run times for this benchmark show that, using adjustable conditions introduce a small overhead which can be ignored. This behavior can be observed for other programs which can be successfully analyzed by all configurations as well and total run times for these configurations verify that the small overhead is compensated by programs which can be analyzed faster by using verification conditions.

For the programs in the benchmark set `ssh`, adjustable assume edges in path condition is the best configuration as it can successfully analyze more programs than other configurations. Explicit analysis using no conditions achieves analyzing 16 programs whereas using it with path length condition or repeating locations in path condition can analyze one more program. The assume edges in path condition can analyze 18 programs and it uses less time than the configuration with no conditions.

Adjustable conditions shows its strength on the benchmark set `systemc`. Explicit analysis using path length condition or assume edges in path condition can analyze 25 programs in that benchmark set and if no conditions are used, 8 of them cannot be successfully analyzed. Using repeating locations in path condition can also analyze 2 more programs than explicit analysis with no conditions. Using adjustable conditions also spends less time than the no conditions configuration for `systemc` benchmark programs.

Comparing threshold values for `pc_sfifo_2_BUG` and `token_ring.14.BUG` demonstrates why it might be useful to use adjustable conditions. We used conditions before with fixed threshold values by guessing a suitable limit for a given program which is shown in Table 5.3. Determining the threshold value by observing some specific properties of the analyzed program might be misleading. Both `pc_sfifo_2_BUG` and `token_ring.14.BUG` times out when we use explicit analysis and the configuration with no conditions does not give any information to improve the analysis. When we use adjustable conditions by restarting the analysis, we use some information from the previous runs of the algorithm. Since we use the same reached set that is generated in the previous run, the analysis is almost as fast as using no

| Program | Only Time Limit 180 s | | Time Limit + Path Length | | | Time Limit + Assume Edges | | |
|---|---|---|---|---|---|---|---|---|
| | Time | $\|R\|$ | Time | Thr. | $\|R\|$ | Time | Thr. | $\|R\|$ |
| `test_locks_13` | 181 | 141416 | 181 | 28 | 207890 | 181 | 8 | 171934 |
| `test_locks_14` | 181 | 116759 | 181 | 30 | 132763 | 181 | 8 | 174988 |
| `test_locks_15` | 181 | 111607 | 180 | 32 | 118827 | 181 | 9 | 179009 |
| `diskperf_simpl1` | 181 | 131955 | 181 | 17425 | 154101 | 181 | 4105 | 159828 |
| `s3_srvr_7` | 181 | 268218 | 181 | 604 | 261975 | 181 | 193 | 261843 |
| `token_ring.10.BUG` | 182 | 441393 | 183 | 792 | 429152 | 182 | 150 | 419782 |
| `token_ring.10` | 190 | 432018 | 181 | 792 | 420721 | 188 | 120 | 430376 |
| `token_ring.11.BUG` | 180 | 376227 | 182 | 924 | 434472 | 189 | 178 | 403326 |
| `token_ring.11` | 182 | 386384 | 181 | 924 | 406315 | 181 | 194 | 392594 |
| `token_ring.12.BUG` | 182 | 340296 | 182 | 915 | 361409 | 182 | 190 | 368210 |
| `token_ring.12` | 181 | 334267 | 183 | 915 | 365917 | 182 | 224 | 353825 |
| `token_ring.13.BUG` | 181 | 328190 | 182 | 1230 | 369330 | 182 | 266 | 369337 |
| `token_ring.13` | 184 | 338923 | 182 | 1312 | 365736 | 181 | 285 | 356297 |
| TOTAL | | 3747653 | | | 4028608 | | | 4041349 |

Table 5.5: Comparison of the size of reached set using adjustable conditions

restarts and we can set a better threshold instead of just making a guess or using a common threshold value for all programs. For example the analysis set 18114 for `pc_sfifo_2_BUG` and 839 for `token_ring.14.BUG` as the threshold value for path length condition which has a huge difference between them. This was only possible by using some information from the analysis itself and adjusting accordingly until a suitable threshold value is found.

### 5.7.2 Adjustable Conditions for Improving Coverage

In Table 5.5, we present the total number of reached states at the end of the analysis for programs which terminated with a time-out in Table 5.4. We used programs that failed to report a useful result after a time-out because other programs that did report UNKNOWN were imprecise due to imprecision of explicit analysis. Since they terminated before reaching to the time limit, and in most cases in a few seconds, they are not good candidates to compare coverage of the state space of the program. The programs that we included in Table 5.5 use the entire time limit allocated for them, but they fail to report a conclusive result. Most of them would time out even when the time limit is increased. The user might still want to use results from a timed-out analysis to check what parts of the state space is explored. Therefore, we analyzed timed-out programs using a time limit and adjustable conditions.

The column 'Program' lists the name of the programs. 'Only Time Limit' column lists results for the configuration with only 180 s time limit. 'Time Limit + Path Length' column

lists results for the configuration with 180 s time limit and adjustable path length condition. Finally, 'Time Limit + Assume Edges' column lists results for the configuration with 180 s time limit and adjustable assume edges condition. We use the *Total Time* condition from Table 5.1 for limiting the time, so that instead of crashing with no results, we terminate gracefully and report the output at the end of the time limit. Some run-times reported are higher than the time-limit due to pre and post-processing times. The columns '$|R|$' show the total number of states in `reached` at the end of the analysis. For adjustable conditions columns, 'Thr' sub-column shows the threshold value when the analysis terminated.

For most programs, using adjustable configurations discovered more states. With only total time condition, analysis for `s3_srvr_7`, `token_ring.10.BUG`, and `token_ring.10` reported more states than the other two configurations. For the rest of the programs, both path length condition and assume edges condition explored more states than the total time limit condition only. For some programs, the difference is as high as 60 %. For example, for `test_locks_15`, assume edges reported 179009 states and only total time configuration reported 111607 states. Similarly, path length reported 434472 abstract states for `token_ring.11.BUG`, whereas only total time configuration reported 376227 states. Indeed, comparing the total number of explored abstract states reveals that configurations that used adjustable conditions explored almost 21000 more states per program in average for the 13 programs shown in the table.

The number of total abstract states reported at the end of the program might be a useful tool to deduct facts about the program with explicit analysis. Explicit analysis continuously builds the ARG by exploring states and the parts of the ARG is never removed or shrank during the analysis. So, if a program cannot terminate with a meaningful result in the given time limits, exploring a bigger part of the state space might be helpful to observe the behaviour of the program.

In conclusion, using adjustable conditions introduces a very small overhead to explicit analysis which can be easily ignored. On the other hand, by using information from previous runs of the analysis, it adjusts and tries a new limiting condition to improve the verification process. Our experiments have shown us that using adjustable conditions can cover a bigger part of the state space and can discover more bugs in the benchmark programs. They are also useful for understanding and studying the behaviour of a program. In combination with assumptions, conditions can be very useful for programs that cannot be analyzed under given resources. Instead of letting the tool crash at the end of a time limit or by consuming the

resources allocated, these components can be limited by a condition to prevent the analysis to reach to a critical state. Upon termination, conditions generate an assumption and in combination with the regular output of the analysis such as proof certificates, set of reached states or error traces, the generated assumption gives information about the work done by the analysis.

## 5.8 Using Output of one Algorithm as Input for another Algorithm

Sequential composition of analyses is shown to be useful in two ways: To restart different analyses one after another to utilize their strengths in order to achieve precise and efficient analysis; and to adjust the conditions under which the verification is run. In this section, we show another mechanism to make the analysis more efficient using sequential composition framework. Different techniques are better at handling different programs and properties as explained and shown in previous chapters. For some programs, it is likely to have a part of the state space which can be handled more efficiently by a particular technique and another part to be handled by another technique more efficiently. In many cases, distinguishing these parts manually and running separate tasks for analyzing such a program is not a feasible task. Using the sequential composition framework, we propose to run an analysis on a program and if the result is UNKNOWN, then try to eliminate parts of the state space that is explored by that analysis. Then, using this information, we can switch to another analysis and try to verify the remaining part of the state space. For example, an analysis might not be precise or efficient enough to analyze a part of the program, but it might already have analyzed other parts of the program. Another analysis can work only on a part that is not explored in full by the first analysis.

### 5.8.1 Example

Let us revisit the example given in Figure 5.2 at the beginning of this chapter. The CFA for the C program with the non-linear relation is given in Figure 5.13. After CFA location 2, the program may follow one of the two branches: The left branch has a loop with a very large index, and the right branch has a condition with a non-linear relation. As explained for this example, the left part of the branch can be handled by predicate analysis easily

given a predicate ($i \geq 1000000$), and the right part of the branch cannot be handled due to the non-linear assignment `r = x * y;`. On the other hand, the explicit analysis can easily handle the right part of the branch using explicit assignments to the variables but cannot handle the left part because it has to unwind the loop one million times before validating the program.

The ARG for the predicate analysis of this program with the only predicate ($i \geq 1000000$) is shown in Figure 5.14. The abstract state associated with the ARG node is shown next to the node. For this example, we assume that we run a precise path analysis after the predicate analysis fails to prove the assertion at line 13, `assert(r >= x);`. CBMC can be used for feasibility check of the error path as explained in Section 3.3.1. The feasibility check will report that the path to the error label is not feasible, thus the tool will report an UNKNOWN answer due to imprecise analysis and the assumption generation algorithm will generate an assumption for the assertion. In the ARG, we show the generated assumptions next to the edges, and the only assumption which is not *true* is between nodes 12 and 13 which is $r \geq x$. So the output of the analysis will be the set `reached` which is equivalent to the ARG shown in Figure 5.14 and the assumption which is $(l = 12) \Rightarrow (r \geq x)$. We can use the output assumption to detect the parts of the ARG which is exhaustively explored and which are yet to be analyzed to have a sound result.

We convert the ARG into an automaton, based on the ARG and the generated assumption. This automaton is called as the *assumption automaton*. The assumption automaton of this example is shown in Figure 5.15. Each transition of the automaton represents the location of the program and the assumption for that location. For example, $9 \rightarrow 10$ with the assumption `TRUE` is the transition from location 9 to 10 (which is the edge `int x = 5;`), and the assumption at this location was *true*. All the nodes at the left part of the ARG have the same assumption *true*. So, they all reach to a final state `T`. The sink state `T` has a self edge that matches any CFA edge. Other transitions that lead to an assumption formula which is different than *true* corresponds one-to-one to the flow of the ARG (the ARG edges).

The assumption automaton gives more information about the progress of an analysis compared to an assumption formula. For instance, the example assumption formula given in Figure 5.11 had two parts which are assumption formulas for the same location. Using an assumption automaton, we can check which transitions were taken on the CFA to reach to each part of the assumption formula. The assumption formula can be used as an input for another analysis as well. Given an assumption automaton as an input, the analysis can run

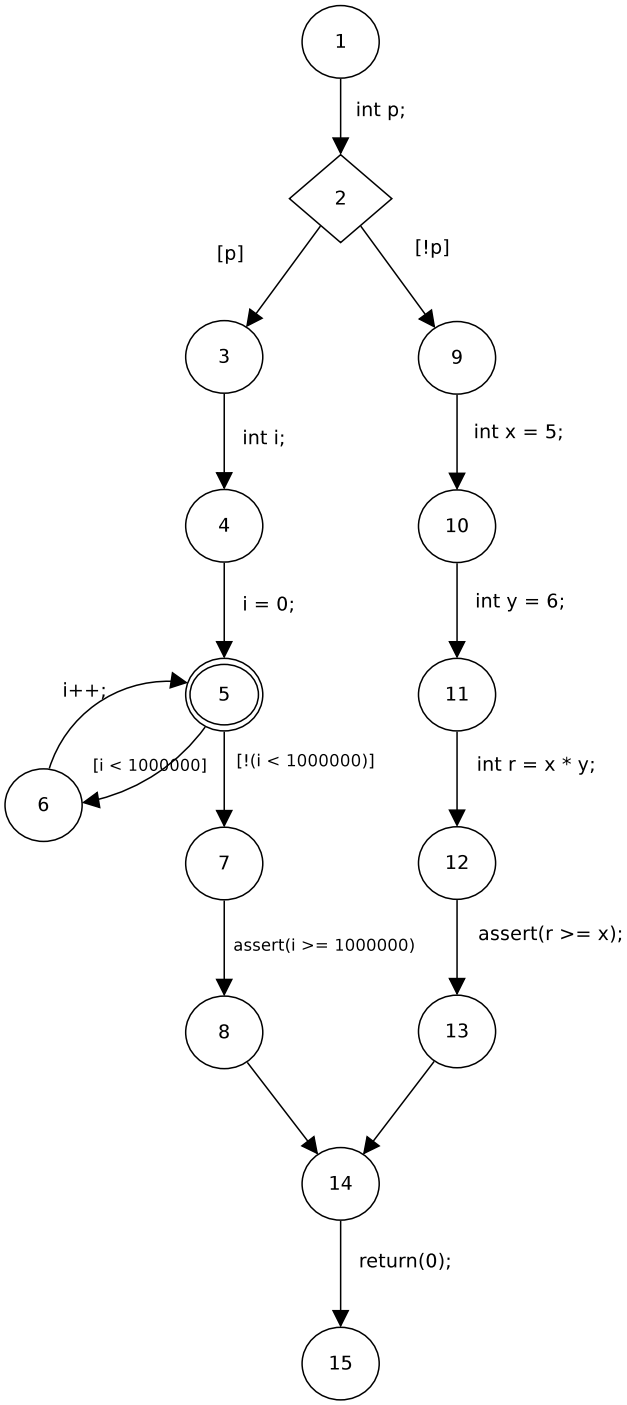


Figure 5.13: CFA for the example program given in Figure 5.2

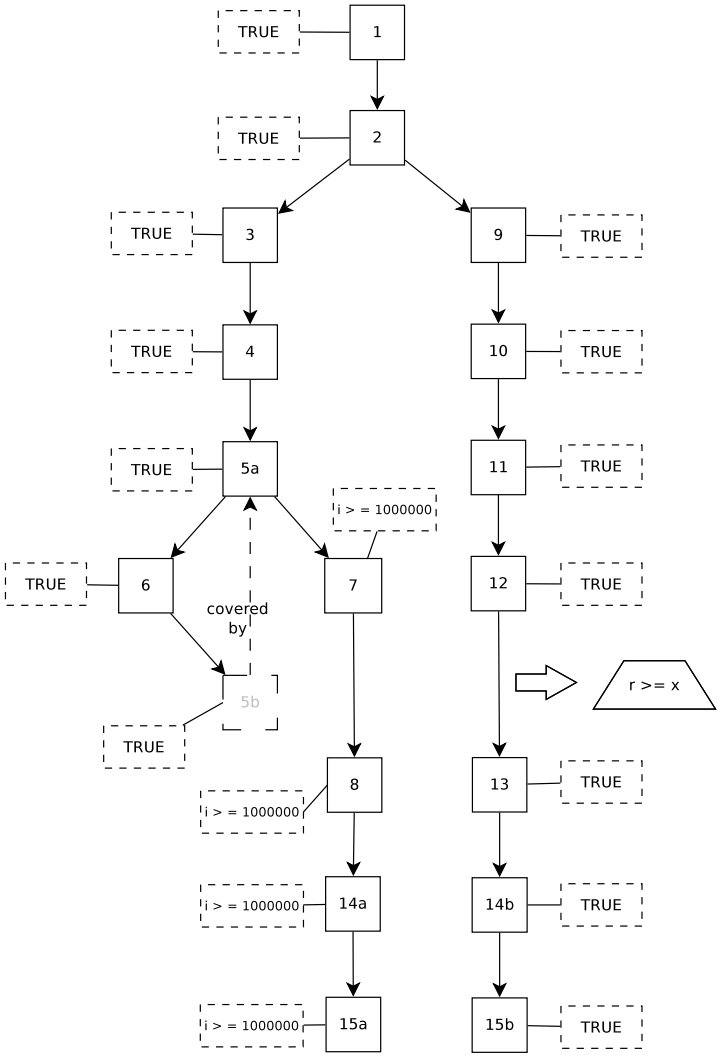


Figure 5.14: ARG for the example program with the CFA in Figure 5.13 after predicate analysis with the predicate ($i \geq 1000000$) followed by a precise path analysis

the automaton in parallel with the analysis and skip the transitions that lead to the sink state (i.e., the transitions that follows a path consists of edges with an assumption that is *true*). For our example with the ARG in Figure 5.14 and with the assumption automaton in Figure 5.15, the left part of the ARG after node 2 is fully discovered. The analysis of the right part is found to be imprecise and the imprecision is captured by the assumption generated for the assertion. The second analysis using the assumption automaton skips the

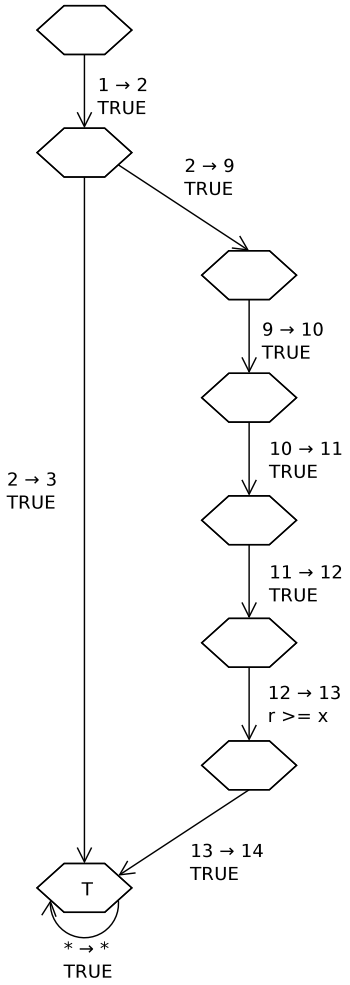


Figure 5.15: Assumption automaton for the example program

parts that collapse to the automaton state `T`. This will ensure that the part that has been already explored will not be analyzed again.

For our example, an explicit analysis can be started using the assumption automaton given in Figure 5.15. Then, it will not go into the loop on the left part of the ARG and it will avoid unwinding it one million times, which would use allocated resources for the analysis. Instead, it will skip the part that has already been analyzed and verified by the first analysis and check the assertion on the right branch of the ARG. Explicit analysis will easily prove that the assertion holds for this path and the analysis will terminate and report SAFE for the program.

## 5.9 Experiments for Using Output of an Algorithm as Input

Explicit analysis can only model explicit assignments to variables, and more complex relations cannot be handled by it. Due to this, it reports many false alarms. In order to decrease the number of false alarms, we check the feasibility of an error path using CBMC as explained in Section 3.3.1. In some cases, the analysis continues after ruling out an infeasible path and eventually finds and reports a real bug. But in many cases, if no real error is found, the explicit analysis cannot report a conclusive answer to the verification problem and an UNKNOWN result is given. Even though the explicit analysis can return no useful answers after analyzing a program, its result can be used to refine another analysis such as predicate analysis.

In Table 5.6, we show the experiments for a combination of analysis with the first analysis' output assumption automaton used as the input to the second algorithm. In the first column, the names of the analyzed programs are shown. The second columns gives the results for explicit analysis with 15 min time limit. The third column is for predicate analysis using LBE approach. The forth column lists the results for the Combination A, explicit analysis with 10 s time limit and predicate analysis from Table 4.1. In the last column, we have the following configuration: We first use explicit analysis with a 5 min time limit, to give it enough time to cover as many states as possible. When the explicit analysis terminates, we build the assumption automaton for the analysis and start predicate analysis with LBE given the assumption automaton as input. We use 10 MB of Java stack for these experiments since the algorithm for building the automaton may require a larger stack than the default value. We used this approach on programs which were not successfully analyzed by CPAchecker in Table 4.1.

As shown in the table, the explicit analysis can analyze 4 of these programs successfully but predicate analysis with LBE or sequential combination of explicit analysis and predicate analysis could not achieve to analyze any of them. Explicit analysis which terminated in a few seconds with an UNKNOWN answer is an indication of imprecise analysis due to its imprecision. So, explicit analysis can run these programs and reach to some errors but since it reports false alarms and cannot find a real bug in the program, the result is imprecise. It already has explored some part of the state space for these programs, so we can use this result by producing an assumption automaton that summarizes the work done. Predicate analysis is then guided using the automaton and it has to work on a much smaller part of

the state space. Predicate analysis can efficiently analyze parts of the state space that the explicit analysis was not able to explore due to imprecision because even if the predicate analysis reaches to an infeasible error, the refinement procedure generates a more precise abstraction.

Results shows a major improvement for the combination of explicit analysis and predicate analysis using the assumption automaton. 21 program can be analyzed which is 17 more than explicit analysis itself and 21 more than predicate analysis or Combination A. Some of them, such as `mem_slave_tlm.1` takes less than 10 s in total, since explicit analysis explored most of the state space but was not able to report a conclusive result due to parts which cannot be handled precisely. This configuration is particularly useful for the programs which explicit analysis can run very efficiently but cannot give a useful result for them. For 28 programs on the table, explicit analysis terminates before reaching the time limit but cannot report SAFE or UNSAFE. For these programs, hundreds of thousands of abstract states are computed but they were not useful since we cannot use the set reached from explicit analysis for running predicate analysis. Converting the entire set reached of explicit analysis to be compatible with predicate analysis would be an impractically expensive operation. So, using assumptions automaton helped us to use this information for another analysis.

These experiments also show us that using a configurable framework like CPAchecker may solve many problems without requiring much effort. Instead of providing a new implementation for using the output of explicit analysis for improving the efficiency of predicate analysis, we simply created a configuration file. This was possible by using the sequential composition framework and configuring it to use a generated automaton for the second algorithm.

## 5.10 An Efficient Combination and Discussion

In Table 5.7, we combine several configurations for CPAchecker to get the best results using sequential composition, verification conditions and verification assumptions.

Figure 5.16 shows how we configured a combination of analyses. We start the analysis with the CBMC with $k = 1$ configuration. Next, we start the verification task with explicit analysis using adjustable assume edges monitoring CPA. We have observed that in many cases, it returns a result in less than 1 min, so we limited it to 1 min. Next, inspired from SATabs's performance on the benchmark set `systemc`, we use the LBE approach with

| Program | Explicit | | Predicate | | Comb. A Expl. + Pred. | | Conditional MC Expl. + Pred. | |
|---|---|---|---|---|---|---|---|---|
| pipeline_BUG | – | 11.0 | – | 900 | – | 900 | – | 197 |
| token_ring.03.BUG | – | 2.60 | – | 900 | – | 900 | ✓ | 5.88 |
| token_ring.04.BUG | – | 2.90 | – | 900 | – | 900 | ✓ | 8.25 |
| token_ring.05.BUG | – | 3.81 | – | 900 | – | 900 | ✓ | 14.4 |
| token_ring.06.BUG | – | 4.94 | – | 900 | – | 900 | ✓ | 23.2 |
| token_ring.07.BUG | – | 8.83 | – | 900 | – | 900 | ✓ | 63.9 |
| token_ring.08.BUG | – | 29.1 | – | 900 | – | 900 | ✓ | 148 |
| token_ring.09.BUG | – | 125 | – | 900 | – | 900 | – | 900 |
| token_ring.10.BUG | – | 501 | – | 900 | – | 900 | – | 363 |
| token_ring.11.BUG | – | 900 | – | 900 | – | 900 | – | 401 |
| token_ring.12.BUG | – | 900 | – | 900 | – | 900 | – | 381 |
| token_ring.13.BUG | – | 900 | – | 900 | – | 900 | – | 405 |
| token_ring.14.BUG | ✓ | 498 | – | 900 | – | 900 | – | 428 |
| token_ring.15.BUG | – | 900 | – | 900 | – | 900 | – | 378 |
| transmitter.10.BUG | ✓ | 28.0 | – | 900 | – | 900 | ✓ | 42.0 |
| transmitter.11.BUG | ✓ | 124 | – | 900 | – | 900 | ✓ | 133 |
| transmitter.12.BUG | ✓ | 501 | – | 900 | – | 900 | – | 377 |
| transmitter.13.BUG | – | 900 | – | 900 | – | 900 | – | 369 |
| mem_slave_tlm.1 | – | 2.17 | – | 900 | – | 900 | ✓ | 4.69 |
| mem_slave_tlm.2 | – | 2.46 | – | 900 | – | 900 | ✓ | 7.22 |
| mem_slave_tlm.3 | – | 2.73 | – | 900 | – | 900 | ✓ | 7.22 |
| mem_slave_tlm.4 | – | 3.18 | – | 900 | – | 900 | ✓ | 8.16 |
| mem_slave_tlm.5 | – | 3.58 | – | 900 | – | 900 | ✓ | 8.91 |
| pipeline | – | 11.2 | – | 900 | – | 900 | – | 265 |
| token_ring.02 | – | 1.82 | – | 900 | – | 900 | ✓ | 3.77 |
| token_ring.03 | – | 2.17 | – | 900 | – | 900 | ✓ | 5.60 |
| token_ring.04 | – | 2.67 | – | 900 | – | 900 | ✓ | 9.96 |
| token_ring.05 | – | 3.66 | – | 900 | – | 900 | ✓ | 18.0 |
| token_ring.06 | – | 5.53 | – | 900 | – | 900 | ✓ | 34.5 |
| token_ring.07 | – | 8.84 | – | 900 | – | 900 | ✓ | 97.8 |
| token_ring.08 | – | 27.4 | – | 900 | – | 900 | ✓ | 397 |
| token_ring.09 | – | 96.3 | – | 900 | – | 900 | – | 900 |
| token_ring.10 | – | 486 | – | 900 | – | 900 | – | 391 |
| token_ring.11 | – | 900 | – | 900 | – | 900 | – | 421 |
| token_ring.12 | – | 900 | – | 900 | – | 900 | – | 470 |
| token_ring.13 | – | 900 | – | 900 | – | 900 | – | 457 |
| toy | – | 2.34 | – | 900 | – | 900 | ✓ | 7.18 |
| **Total** | 4 | 9700 | – | 33300 | – | 33300 | 21 | 8150 |

Table 5.6: Experiments with assumption automata as condition

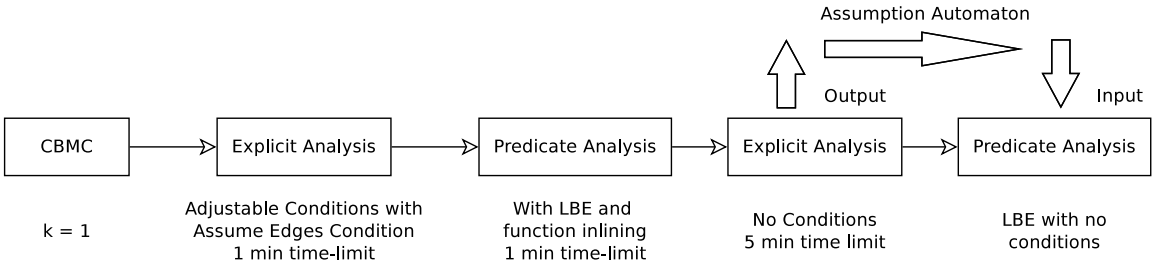


Figure 5.16: Flow of the configurations

function inlining, but to avoid memory crashes, we limit it only with 1 min hoping that it will successfully analyze some programs such as `pipeline` in a short time without reaching to critical memory usage. The remaining time is allocated for our approach with assumption automaton. We run the explicit analysis with no conditions for 5 min to allow it to discover some parts of the state space entirely in depth and then we use the resulting assumption automaton for the standard LBE approach with no conditions.

The total time limit for all examples is again set to 15 min. Total memory limit is 14 GB, Java heap limit is 4 GB and Java stack size is 10 MB. The sequential combinations of several optimized configurations shows a dramatic improvement. We can analyze 97 programs successfully, significantly more than any tools or configurations that we used for experiments so far. The only program that is not analyzed with this configuration is `pc_sfifo_2`, even though single configurations were able to analyze it. This is due to the difficulty of monitoring heavy-load calls to external programs. For some programs, calling CBMC or MathSAT as an external process, the tool cannot terminate them gracefully in the middle of their execution using a time-limit. A similar issue is discussed in Chapter 6 as future work. These results' significance comes from their flexibility. Combining conditions, assumptions and different analysis techniques gives the user greater flexibility of combining different approaches. The user does not have to worry about implementation details for each analysis used to optimize their strengths, but by limiting the resources or state space search with configurable conditions and combining them sequentially, the performance of the tool has been improved dramatically.

Table 5.7: Combinations of 5 different configurations

| Program Name | Result | *Time* |
|---|---|---|
| `test_locks_14.BUG` | ✓ | 1.37 |
| `test_locks_15.BUG` | ✓ | 1.40 |
| `test_locks_5` | ✓ | 2.03 |
| `test_locks_6` | ✓ | 2.35 |
| `test_locks_7` | ✓ | 2.87 |
| `test_locks_8` | ✓ | 3.52 |
| `test_locks_9` | ✓ | 5.23 |
| `test_locks_10` | ✓ | 9.30 |
| `test_locks_11` | ✓ | 64.9 |
| `test_locks_12` | ✓ | 64.5 |
| `test_locks_13` | ✓ | 63.5 |
| `test_locks_14` | ✓ | 64.7 |
| `test_locks_15` | ✓ | 64.5 |
| **Locks total** | 13 | 350 |
| `cdaudio_simpl1_BUG` | ✓ | 2.82 |
| `floppy_simpl3_BUG` | ✓ | 1.75 |
| `floppy_simpl4_BUG` | ✓ | 1.96 |
| `kbfiltr_simpl2_BUG` | ✓ | 1.68 |
| `cdaudio_simpl1` | ✓ | 2.84 |
| `diskperf_simpl1` | ✓ | 65.9 |
| `floppy_simpl3` | ✓ | 1.63 |
| `floppy_simpl4` | ✓ | 1.93 |
| `kbfiltr_simpl1` | ✓ | 1.53 |
| `kbfiltr_simpl2` | ✓ | 1.55 |
| **NT drivers total** | 10 | 83.6 |
| `s3_clnt_1_BUG` | ✓ | 5.74 |
| `s3_clnt_2_BUG` | ✓ | 5.79 |
| `s3_clnt_3_BUG` | ✓ | 5.77 |

Continued on next page

**Table 5.7 – continued from previous page**

| Program Name | Result | *Time* |
|---|---|---|
| s3_clnt_4_BUG | ✓ | 6.08 |
| s3_srvr_11_BUG | ✓ | 2.84 |
| s3_srvr_12_BUG | ✓ | 2.84 |
| s3_srvr_14_BUG | ✓ | 3.62 |
| s3_srvr_1_BUG | ✓ | 2.22 |
| s3_srvr_2_BUG | ✓ | 2.33 |
| s3_srvr_6_BUG | ✓ | 1.72 |
| s3_clnt_1 | ✓ | 13.3 |
| s3_clnt_2 | ✓ | 12.8 |
| s3_clnt_3 | ✓ | 13.7 |
| s3_clnt_4 | ✓ | 12.8 |
| s3_srvr_1 | ✓ | 3.65 |
| s3_srvr_1a | ✓ | 4.98 |
| s3_srvr_1b | ✓ | 2.34 |
| s3_srvr_2 | ✓ | 3.55 |
| s3_srvr_3 | ✓ | 18.4 |
| s3_srvr_4 | ✓ | 9.60 |
| s3_srvr_6 | ✓ | 523 |
| s3_srvr_7 | ✓ | 86.3 |
| s3_srvr_8 | ✓ | 3.65 |
| **SSH total** | 23 | 747 |
| kundu1_BUG | ✓ | 2.43 |
| kundu2_BUG | ✓ | 2.01 |
| pc_sfifo_1_BUG | ✓ | 34.5 |
| pc_sfifo_2_BUG | ✓ | 62.9 |
| pipeline_BUG | ✓ | 27.1 |
| token_ring.01.BUG | ✓ | 3.23 |
| token_ring.02.BUG | ✓ | 5.25 |
| token_ring.03.BUG | ✓ | 7.69 |

Continued on next page

**Table 5.7 – continued from previous page**

| Program Name | Result | *Time* |
|---|---|---|
| token_ring.04.BUG | ✓ | 16.5 |
| token_ring.05.BUG | ✓ | 76.3 |
| token_ring.06.BUG | ✓ | 87.3 |
| token_ring.07.BUG | ✓ | 125 |
| token_ring.08.BUG | ✓ | 96.9 |
| token_ring.09.BUG | - | 673 |
| token_ring.10.BUG | - | 900 |
| token_ring.11.BUG | - | 900 |
| token_ring.12.BUG | - | 471 |
| token_ring.13.BUG | - | 900 |
| token_ring.14.BUG | ✓ | 38.3 |
| token_ring.15.BUG | ✓ | 40.6 |
| toy1_BUG | ✓ | 66.7 |
| toy2_BUG | ✓ | 2.44 |
| transmitter.01.BUG | ✓ | 1.85 |
| transmitter.02.BUG | ✓ | 2.50 |
| transmitter.03.BUG | ✓ | 2.15 |
| transmitter.04.BUG | ✓ | 2.48 |
| transmitter.05.BUG | ✓ | 2.98 |
| transmitter.06.BUG | ✓ | 3.72 |
| transmitter.07.BUG | ✓ | 4.62 |
| transmitter.08.BUG | ✓ | 6.81 |
| transmitter.09.BUG | ✓ | 13.9 |
| transmitter.10.BUG | ✓ | 38.3 |
| transmitter.11.BUG | ✓ | 37.1 |
| transmitter.12.BUG | ✓ | 40.0 |
| transmitter.13.BUG | ✓ | 39.3 |
| transmitter.15.BUG | ✓ | 18.5 |
| transmitter.16.BUG | ✓ | 18.7 |
| bist_cell | ✓ | 1.60 |

Continued on next page

**Table 5.7 – continued from previous page**

| Program Name | Result | *Time* |
|---|---|---|
| `kundu` | ✓ | 10.1 |
| `mem_slave_tlm.1` | ✓ | 68.9 |
| `mem_slave_tlm.2` | ✓ | 71.7 |
| `mem_slave_tlm.3` | ✓ | 85.0 |
| `mem_slave_tlm.4` | ✓ | 128 |
| `mem_slave_tlm.5` | ✓ | 323 |
| `pc_sfifo_1` | ✓ | 455 |
| `pc_sfifo_2` | - | 900 |
| `pc_sfifo_3` | ✓ | 2.27 |
| `pipeline` | ✓ | 29.2 |
| `token_ring.01` | ✓ | 4.14 |
| `token_ring.02` | ✓ | 6.78 |
| `token_ring.03` | ✓ | 32.0 |
| `token_ring.04` | ✓ | 69.4 |
| `token_ring.05` | ✓ | 79.7 |
| `token_ring.06` | ✓ | 105 |
| `token_ring.07` | ✓ | 165 |
| `token_ring.08` | ✓ | 446 |
| `token_ring.09` | - | 900 |
| `token_ring.10` | - | 900 |
| `token_ring.11` | - | 450 |
| `token_ring.12` | - | 416 |
| `token_ring.13` | - | 514 |
| `toy` | ✓ | 67.6 |
| **SystemC total** | 51 | 10900 |
| **Total** | 97 | 12100 |

## 5.11 Related Work

Software verification tools aim to report the best result in an efficient way. The best result can be defined as reporting a genuine bug in the case that the software contains a bug or producing a 'convincing' safety certification otherwise. Comprehensive analysis of a program is impractical in many cases due to large size of many software systems. Almost all software analysis tools make sacrifices to be practical, such as being partially sound instead of guaranteeing total soundness of the analysis. A formal system should clearly define under which assumptions the tool defines soundness to be practical [60].

Existence of infinite paths presents a challenging situation. Bounding path length for symbolic execution is proposed from very early stages of the development of symbolic execution [85]. For example, bounding is used for analyzing dynamic data structures by limiting the number of heap cells considered, so that SAT solvers can handle the formula efficiently [80]. Bounded model checking is also very successful at finding bugs efficiently by limiting the number of unwinding of loops in the program [44, 47]. This limitation makes bounded model checking sound under the assumption that loops are unrolled a specific number of times. Verification tools that use heuristics search might follow a similar approach to handle large-scale programs efficiently [60].

Model checker TCP tries to run the analysis by selecting a different state based on some heuristics and claims to explore as many states as possible to detect a bug before resources are consumed [94]. One of the heuristics used by TCP is to eliminate a variable when it takes a value more than a certain number of times on a path which is also used by dynamic precision adjustment approach to delegate handling of such a variable to a different static analysis [25]. Our work is similar to heuristics-based search by incorporating conditions based on explored region of the state space. We also let the user to bound infinite paths or drop some states in order to proceed with more interesting paths and cover a bigger part of the state space.

The assume-guarantee paradigm is used for a long time for all types of software analysis purposes [72]. Following a similar pattern to assume-guarantee reasoning, ESC/Java uses an annotation language for Java to encode assumptions for pruning false alarms [61]. It is a static analysis tool which can detect many properties such as null dereferencing, array bounds errors etc. but the developer can use an annotation language in cases where he is sure that the code is safe. They claim that the use of ESC/Java lets the developer

to choose the optimum point between soundness and being practical. Nimmer and Ernst proposed an approach to extract program specifications from dynamic analysis [98]. Their approach make use of imprecise analysis to improve the efficiency and soundness of static analysis. They show that generated specifications can cover a big part of the state space very efficiently.

In a recent work, Conway et al. defines conditional soundness for points-to analysis [50]. It is similar to our definition such that it requires abstract states to be feasible with respect to a predicate. The analysis is said to be sound under that predicate. They also note that their definition can be extended for static analysis but they did not present any cases where they used their approach in practice. It should also be noted that well-known verification tools such as SLAM, ESC/Java, Blast and the tool that we use, CPAchecker, already runs under the assumption that the program does not contain integer overflows, or floating point errors [11, 18]. Tools like CBMC, on the other hand, can detect some of these errors because it supports bit-wise operations [44].

JPF is a tool which allows arbitrary user-defined search strategies [64, 65]. Groce and Visser uses heuristics to run model checking efficiently on multi-threaded Java programs. They experiment with different search strategies and propose new heuristics based on the structure of the program such as branching or thread-independence. They also used a static date-race detection algorithm and using the potential races, they generated a thread preference for the search strategy [65]. Mur$\varphi$++ uses heuristics to guide the search strategy during model checking [108]. The strategy uses an approach that uses a distance-based metric which is measured by comparing similarities of states. They also try to use an incremental model checking approach to increase the chance of detecting a bug.

We present a general way of working with unhandled cases instead of giving up the analysis by a crash or ignoring the case altogether. Since we work on a flexible software analysis framework, our approach can be easily used with any static analyses, or even combinations of static analyses. Our approach can use similar techniques to heuristics search as well as techniques that use assumptions and user annotations through the invariant language and automaton.

# Chapter 6

# Discussion

We conclude the thesis by giving a summary of the state of software verification research and analysis tools. A summary of our work is given and we discuss how it may be placed in the current state of the verification research.

## 6.1 State-of-the-Art in Software Verification

Hoare describes 'The Verifying Compiler' as one of the grand challenges for computer science research [75, 76]. The verifying compiler checks the correctness of the program that it compiles and guarantees that its behaviour is correct. He claims that the development of such a tool does not only help in checking the correctness, but it will also have positive impacts on the development and maintenance stages of software systems. He lists many tools that are used in software verification that have shown great improvements in recent years, such as model checking or SAT solving [92, 95]. He notes that combining these tools is one of the challenges of ongoing research:

> "A major remaining challenge is to find effective ways of combining this wide range of component technologies into a small number of tools, to meet the needs of program verification." [...]

Discussing the developments in the verification tools for some specific cases, he adds that a general, widely-accepted approach is still one of the biggest challenges. He concludes his discussion by acknowledging that the ultimate goal of the verification research and current contributions is constructing reliable software:

> "Much of its [verifying compiler's] use may be confined to the relatively lower levels of verification. It is a common fate of grand challenges that achievement of their direct goal turns out to be less directly significant than the stimulus that its pursuit has given to the progress of science and engineering. But remember, that was the primary purpose of the whole exercise." [...]

Alglave et al. discuss the state of verification tools and their practicality [1]. They note that the tool development in software verification research is not fully acknowledged and more credit should be given to the development of robust and reliable tools. Given that one major burden on software verification research is being accepted and used by industry for real purposes in a wide scale, research groups that devote significant time on developing software verification tools should get more support and encouragement from the research community. As an important example, the development of SLAM is given. It contributed immensely to developments and improvements in predicate abstraction and CEGAR approaches. The major contribution of SLAM was due to its development purpose: To verify industrial code. It is used for testing Windows device drivers and serves as the engine of the Windows 7 Driver Kit [1] [5, 7, 10]. ASTRÉE is also being used in the industry, especially for checking correctness of Airbus flight control software [2] [32, 57]. However, the main problem is still having tools that target specific types of programs and properties [1].

Alglave et al. also emphasized the problem of incompatibility and incomparability of tools and techniques. They use recent improvements in SMT the and SAT solving research community as an example to show the value of shared tools and benchmarks. For them, being practical and working at least on a subset of programs that contain a minimal set of properties is essential for a software analysis tool to be functional.

## 6.2 Conclusion

The CPA concept is a step towards combining different analyses [24]. Using different techniques within the CPA framework and using intermediate settings is beneficial in comparing different methods [103]. CPAchecker, which implements a one-to-one representation of the CPA framework, is a step forward in providing a tool that can combine different techniques

[1] http://research.microsoft.com/en-us/projects/slam.

[2] http://www.astree.ens.fr.

and tools within a single tool together. CPAchecker aims to be a tool that is easy to extend and being used by researchers to implement new ideas and techniques easily [27]. To achieve practicality, CPAchecker integrates most break-through technologies from software verification: counterexample checking, counterexample-guided abstraction refinement technique, interfaces to access several SMT solvers and interpolation solvers, several static analyses and model checking algorithms, a parser for the C language to convert it to a control flow automata, an automaton based property specification and interfaces to access commonly used data structure such as BDDs. The tool is used by several research groups for implementing their ideas such as test case generation, analysis of Linux drivers, and for checking correctness of interactions of Internet applications [37, 77].

Currently, we are still working on the development of the tool. CPAchecker will be used for analysis of more types of programs, with extensions that support bit-wise operations, bounded model checking, CEGAR for explicit analysis, analysis of concurrent programs, path invariants for counterexample analysis and better handling of pointers. Adding more features to CPAchecker will help it achieve its goal of being extendable, compatible and practical. Combining different tools is a major challenge of software verification and we hope that CPAchecker will inspire researchers to collaborate and develop robust tools.

### 6.2.1 Contributions

In this thesis, we introduced new techniques and approaches for contributing to configurable interactions and practicality of CPAchecker [19,20]. The main idea behind the CPA concept and dynamic precision was to use several analyses together and configure them together to achieve better efficiency and precision. We extended this definition to use several analyses to be used sequentially and we implemented a framework that can easily use this idea by simply providing configurations.

We used verification conditions and verification assumptions to handle imprecise results of the model checking process. Verification conditions can be used in a similar manner with heuristics to optimize the analysis process. We have provided several heuristics which are externally visible to the user and can be combined with any type of CPA implemented in CPAchecker. Adjustable conditions are particularly useful for types of analyses that discover abstract states one after another and continuously expands the abstract reachability tree. CPAchecker observes the progress of the analysis and identifies parts of the state space

that might be more feasible to discover at the moment. Evaluating the statistics from these observations, the provided condition is adjusted to produce better results for the analysis.

A more direct effort to handle imprecise results was verification assumptions [19, 20]. In order to utilize the effort spent by a tool that does not terminate or to manage imprecise results, we proposed a framework that uses verification assumptions. When the analysis terminates, we generate an assumption for the analysis such that the results were sound under the reported assumption. Even the trivial verification conditions such as conditions that limits time usage or memory usage become efficient tools when combined with assumption collecting algorithm. When the analysis reaches to critical limits of resource usage, the analysis can terminate without crashing and reports its results with generated assumptions. The conditional model checking, combining model checking with conditions and assumptions, minimizes the number of 'fails' this way. Another benefit of verification assumptions was their ability to represent outputs of any analyses using a common formalism. This capability of assumptions increases the flexibly of CPAchecker in combining different analyses together.

Our experiments show that CPAchecker is a great improvement over the state-of-the-art in software model checking. Using sequential composition framework has already improved the process by analyzing more programs successfully in less time. Combining verification conditions and assumptions with other CPAs and with sequential composition, we have analyzed even more programs successfully.

We believe that using a configurable platform for software analysis is crucial for achieving the task of being practical and for bringing different tools together under a shared platform. With CPAchecker, we provided a platform that is configurable and easy to extend. We also provided mechanisms to combine and use several techniques (even different tools) efficiently. CPAchecker is an exciting project that gains more attention from research community and we hope that with this thesis, we contributed to its development by making it more practical and usable.

## 6.3 Future Work

In order to achieve its goals of being a practical software analysis platform, CPAchecker can be extended in several ways.

1. Integration of efficient open-source SMT would provide more flexibility to developers. Currently, we use MathSAT which is a very efficient SMT solver but it is not an open-source project. Since we have to call it as an external tool, we have limited options to control it once it starts solving a problem during analysis. For example, conditions that we implemented which monitors the transfer relation are most useful for analyses that runs heavy tasks. Predicate analysis is a typical example where a huge formula might be built for abstraction and the SMT solver might not be able to generate the models in a reasonable time. In the current implementation, if we force the SMT solver to stop in the middle of the computation, it usually crashes the tool. An open source solver would be easy to be stopped gracefully and we would be able to rule out states which require huge resources to be computed from the analysis using verification conditions.

2. Precision adjustment is also closely related with the concept of verification assumptions. For analyses that implement precision adjustment functions, it determines how the elements from the abstract domain should be used. For example, for numerical analysis, precision may determine the variables that shall be tracked at a location. This suggests that it sometimes loose its precision. Combining dynamic precision adjustment with conditions, a user might adjust the precision based on the statistics from a condition. This way, based on the properties of a discovered path, the dynamic precision may continue the analysis with a CPA and stop the other CPA to track variables.

3. For adjusting the thresholds for conditions, we use statistics such as the length of a path, the time spent for computing a state or the number of occurrences of some certain types of edges during the analysis. More complicated techniques can be investigated in order to come up with more efficient thresholds. For example, statistics based on occurrences of variables can be used. Another approach might be to focus on parts that are more likely to contain a real error. One idea is observing the refinement process of predicate analysis and if the analysis cannot terminate, concentrating on error labels that are visited most frequently using another CPA such as CPA for explicit analysis.

# Bibliography


[1] Jade Alglave, Alastair F. Donaldson, Daniel Kroening, and Michael Tautschnig. Making software verification tools really work. In *Proc. ATVA*, LNCS. Springer, 2011. Invited paper, to appear.

[2] Domagoj Babic and Alan J. Hu. Structural abstraction of software verification conditions. In *Proc. CAV*, pages 366–378, 2007.

[3] Domagoj Babic and Alan J. Hu. Calysto: Scalable and precise extended static checking. In *Proc. ICSE*, pages 211–220, 2008.

[4] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[5] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proc. EuroSys*, pages 73–85, 2006.

[6] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: Static driver verification with under 4% false alarms. In *Proc. FMCAD*, pages 35–42, 2010.

[7] Thomas Ball, Ella Bounimova, Vladimir Levin, Rahul Kumar, and Jakob Lichtenberg. The static driver verifier research platform. In *Proc. CAV*, pages 119–122, 2010.

[8] Thomas Ball, Byron Cook, Satyaki Das, and Sriram K. Rajamani. Refining approximations in software predicate abstraction. In *Proc. TACAS*, pages 388–403, 2004.

[9] Thomas Ball, Byron Cook, Shuvendu K. Lahiri, and Lintao Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Proc. CAV*, pages 457–461, 2004.

[10] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[11] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. PLDI*, pages 203–213, 2001.

[12] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. *STTT*, 5(1):49–58, 2003.

[13] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. SPIN*, pages 113–130, 2000.

[14] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proc. PASTE*, pages 97–103, 2001.

[15] Thomas Ball and Sriram K. Rajamani. Generating Abstract Explanations of Spurious Counterexamples in C Programs, 2002.

[16] Dirk Beyer, Adam Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proc. ICSE*, pages 326–335, 2004.

[17] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *Proc. FMCAD*, pages 25–32, 2009.

[18] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

[19] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional Model Checking. Technical Report MIP-1107, University of Passau, 2011.

[20] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking: A technique to pass information between verifiers. In *Proc. FSE*. ACM, 2012.

[21] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *Proc. VMCAI*, pages 378–394, 2007.

[22] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *Proc. PLDI*, pages 300–309, 2007.

[23] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy shape analysis. In *Proc. Software Verification: Infinite-State Model Checking and Static Program Analysis*, 2006.

[24] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV*, pages 504–518, 2007.

[25] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*, pages 29–38, 2008.

[26] Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz, and Damien Zufferey. Shape refinement through explicit heap analysis. In *Proc. FASE*, pages 263–277, 2010.

[27] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, pages 184–190, 2011.

[28] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197, 2010.

[29] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. CSIsat: Interpolation for LA+EUF. In *Proc. CAV*, pages 304–308, 2008.

[30] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proc. TACAS*, pages 193–207, 1999.

[31] *BLAST User's Manual*, 2005.

[32] Olivier Bouissou, Éric Conquet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Khalil Ghorbal, Éric Goubault, David Lesens, Laurent Mauborgne, Antoine Miné, Sylvie Putot, Xavier Rival, and Michel Turin. Space software validation using abstract interpretation. In *Proc. DASIA*, pages 1–7, 2009.

[33] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proc. ISSTA*, pages 123–133, 2002.

[34] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In *Proc. CAV*, pages 299–303, 2008.

[35] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[36] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proc. SIGPLAN Symposium on Compiler Construction*, pages 152–161, 1986.

[37] André Vicente Calvinho and Rui Gustavo Crespo. Email fi identification and resolution with model checking. *J. Network and Computer Applications*, 34(4):1441–1446, 2011.

[38] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *Proc. TACAS*, pages 397–412, 2008.

[39] Alessandro Cimatti, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Verifying SystemC: A software model checking approach. In *Proc. FMCAD*, pages 51–59, 2010.

[40] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. In *Proc. SAS*, pages 312–327, 2004.

[41] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proc. Logic of Programs*, pages 52–71, 1981.

[42] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, pages 154–169, 2000.

[43] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[44] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, pages 168–176, 2004.

[45] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[46] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS*, pages 570–574, 2005.

[47] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Proc. DAC*, pages 368–371, 2003.

[48] Edmund M. Clarke, Robert P. Kurshan, and Helmut Veith. The localization reduction and counterexample-guided abstraction refinement. In *Proc. Essays in Memory of Amir Pnueli*, pages 61–71, 2010.

[49] Michael Codish, Anne Mulkers, Maurice Bruynooghe, Maria J. García de la Banda, and Manuel V. Hermenegildo. Improving abstract interpretations by combining domains. In *Proc. PEPM*, pages 194–205, 1993.

[50] Christopher L. Conway, Dennis Dams, Kedar S. Namjoshi, and Clark Barrett. Pointer analysis, conditional soundness, and proving the absence of errors. In *Proc. SAS*, pages 62–77, 2008.

[51] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL*, pages 238–252, 1977.

[52] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The Astreé analyzer. In *Proc. ESOP*, pages 21–30, 2005.

[53] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the ASTRÉE static analyzer. In *Proc. ASIAN*, pages 272–300, 2006.

[54] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of static analyzers: A comparison with Astree. In *Proc. TASE*, pages 3–20, 2007.

[55] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL*, pages 84–96, 1978.

[56] William Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.

[57] David Delmas and Jean Souyris. Astrée: From research to industry. In *Proc. SAS*, pages 437–451, 2007.

[58] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[59] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[60] Matthew B. Dwyer, John Hatcliff, Robby, Corina S. Pasareanu, and Willem Visser. Formal software analysis emerging trends in software model checking. In *Proc. FOSE*, pages 120–136, 2007.

[61] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245, 2002.

[62] Robert W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proc. Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia*, volume 19, pages 19–31, Providence RI, 1967. American Mathematical Society.

[63] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proc. CAV*, pages 72–83, 1997.

[64] Alex Groce and Willem Visser. Model checking Java programs using structural heuristics. In *Proc. ISSTA*, pages 12–21, 2002.

[65] Alex Groce and Willem Visser. Heuristics for model checking Java programs. *STTT*, 6(4):260–276, 2004.

[66] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *Proc. PLDI*, pages 376–386, 2006.

[67] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[68] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244, 2004.

[69] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proc. CAV*, pages 526–538, 2002.

[70] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A. A. Sanvido. Extreme model checking. In *Proc. Verification: Theory and Practice*, pages 332–358, 2003.

[71] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70, 2002.

[72] Thomas A. Henzinger, Marius Minea, and Vinayak S. Prabhu. Assume-guarantee reasoning for hierarchical hybrid systems. In *Proc. HSCC*, pages 275–290, 2001.

[73] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. PASTE*, pages 54–61, 2001.

[74] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[75] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. In *Proc. CC*, pages 262–272, 2003.

[76] C. A. R. Hoare, Jayadev Misra, Gary T. Leavens, and Natarajan Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv.*, 41(4), 2009.

[77] Andreas Holzer, Michael Tautschnig, Christian Schallhart, and Helmut Veith. An introduction to test specification in FQL. In *Proc. HVC*, pages 9–22, 2010.

[78] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[79] *IEEE 1666: SystemC language Reference Manual*, 2005.

[80] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proc. ISSTA*, pages 14–25, 2000.

[81] Himanshu Jain, Franjo Ivancic, Aarti Gupta, Ilya Shlyakhter, and Chao Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *Proc. CAV*, pages 137–151, 2006.

[82] Ranjit Jhala and Rupak Majumdar. Path slicing. In *Proc. PLDI*, pages 38–47, 2005.

[83] S. C. Johnson. Lint, a C program checker. In *Proc. CSTR*, pages 78–1273, 1978.

[84] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS*, pages 553–568, 2003.

[85] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[86] Sudipta Kundu, Malay K. Ganai, and Rajesh Gupta. Partial order reduction for scalable testing of SystemC TLM designs. In *Proc. DAC*, pages 936–941, 2008.

[87] Robert P. Kurshan. Verification technology transfer. In *Proc. 25 Years of Model Checking*, pages 46–64, 2008.

[88] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT techniques for fast predicate abstraction. In *Proc. CAV*, pages 424–437, 2006.

[89] William Landi. Undecidability of static analysis. *LOPLAS*, 1(4):323–337, 1992.

[90] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *Proc. PADO*, pages 155–172, 2001.

[91] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[92] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC*, pages 530–535, 2001.

[93] Matthieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, 2005.

[94] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proc. NSDI*, pages 155–168, 2004.

[95] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proc. OSDI*, 2002.

[96] George C. Necula. Proof-carrying code. In *Proc. POPL*, pages 106–119, 1997.

[97] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC*, pages 213–228, 2002.

[98] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proc. ISSTA*, pages 229–239, 2002.

[99] Hendrik Post, Carsten Sinz, Alexander Kaiser, and Thomas Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. In *Proc. ASE*, pages 188–197, 2008.

[100] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[101] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proc. POPL*, pages 38–48, 1998.

[102] Bernhard Steffen. Data flow analysis as model checking. In *Proc. TACS*, pages 346–365, 1991.

[103] Grégory Théoduloz. *Software Verification by Combining Program Analyses of Adjustable Precision.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, Switzerland, 2010.

[104] Willem Visser and Peter C. Mehlitz. Model checking programs with Java Pathfinder. In *Proc. SPIN*, page 27, 2005.

[105] Alexander von Rhein. Verification tasks for software model checking. Master's thesis, Universität Passau, Germany, 2010.

[106] Philipp Wendler. Software verification based on adjustable large-block encoding. Master's thesis, Universität Passau, Germany, 2010.

[107] Yichen Xie and Alexander Aiken. Scalable error detection using boolean satisfiability. In *Proc. POPL*, pages 351–363, 2005.

[108] C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Proc. DAC*, pages 599–604. ACM, 1998.