박 사 학 위 논 문
Ph.D. Dissertation

빠른 추상 도달가능성 분석을 위한 새로운 목표 지향 모델 체킹

A Novel Target Directed Model Checking for Fast Abstract Reachability Analysis

2022

이 낙 원 (李 樂 遠 Lee, Nakwon)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

박 사 학 위 논 문

빠른 추상 도달가능성 분석을 위한 새로운 목표 지향 모델 체킹

2022

이 낙 원

한 국 과 학 기 술 원

전산학부

빠른 추상 도달가능성 분석을 위한 새로운 목표 지향 모델 체킹

이 낙 원

위 논문은 한국과학기술원 박사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2021년 12월 10일

A Novel Target Directed Model Checking for Fast Abstract Reachability Analysis

Nakwon Lee

Major Advisor: Jongmoon Baik
Co-Advisor: Moonzoo Kim

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Daejeon, Korea
December 10, 2021

Approved by

Jongmoon Baik
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

DCS 이낙원. 빠른 추상 도달가능성 분석을 위한 새로운 목표 지향 모델 체킹. 전산학부 . 2022년. 63+v 쪽. 지도교수: 백종문, 김문주. (영문 논문)
Nakwon Lee. A Novel Target Directed Model Checking for Fast Abstract Reachability Analysis. School of Computing . 2022. 63+v pages. Advisor: Jongmoon Baik, Moonzoo Kim. (Text in English)

초 록

추상 도달가능성 기반 모델 체킹 (ARMC)는 상태 폭발 문제를 피하기 위해 실제 프로그램 상태 공간을 추상화한 추상 상태 공간에서 도달가능성 분석을 수행한다. 그런데 기존 추상 도달가능성 기반 모델 체킹 기법들은 목표 경로에 도달하는데 오래 걸리는 문제가 있다. 본 연구에서는 목표 경로를 빨리 발견하여 ARMC의 결함 발견 능력과 검증 속도를 향상시키기 위해 TOUR라는 새로운 기법을 제안한다. TOUR는 실행 시간에 함수 호출 맥락과 에러 위치까지의 거리를 이용해서 에러 위치 지향 탐색을 수행하는 기법이다. TOUR는 네 가지 서로 다른 거리 기준을 이용하며 프로그램의 정적 특성들을 기반으로 거리 기준을 선택할 수 있다. 본 연구에서는 3,042개의 실제 C 프로그램들을 대상으로 TOUR를 엄밀하게 평가한다. 실험 결과 TOUR는 354개의 결함을 가진 프로그램들에 대해서 최신 ARMC 기법보다 20% 많은 프로그램에서 11% 적은 시간 만에 결함을 찾았다. 또한 TOUR는 652개의 복잡한 결함이 없는 프로그램들에 대해서 최신 ARMC 기법보다 15% 많은 프로그램들을 15% 적은 시간 만에 검증했다.

핵 심 낱 말 소프트웨어 검증, 소프트웨어 시험, 심볼릭 모델 체킹, 추상 도달가능성, 함수간 분석, 지향 탐색

Abstract

Reachability analysis is an approach to ensure software reliability by proving that a program cannot reach an unsafe state (e.g., an error location). Abstract Reachabiity-based Model Checking (ARMC) techniques conduct a reachability analysis in an abstraction of a program to mitigate the state-explosion problem. However, existing ARMC techniques are slow to detect a target path (i.e., a path reaching an error location) due to an inefficient search method. We propose a novel technique TOUR (effective error lOcation directed search via an interprocedUral Runtime distance calculation) to improve both bug detection ability and verification speed of ARMC by detecting a target path quickly. The key idea of TOUR is an error location directed search that utilizes the distance to an error location and function call context at runtime. TOUR applies four different distance metrics and a model for program-specific distance metric selection using static features of a program. We have extensively evaluated TOUR on 3,042 real-world C programs in a software verification competition benchmark. The experiment results show that TOUR, due to its error location directed search, finds bugs in 20% more programs in 11% less model checking time than the state-of-the-art ARMC technique (i.e., block-abstraction memoization) for 354 buggy programs. Also, TOUR verifies 15% more programs within 15% less model checking time than the block-abstraction memoization for 652 complex clean programs. In conclusion, TOUR achieves fast reachability analysis by speeding up ARMC, and the fast reachability analysis helps practitioners to efficient software reliability assurance.

Keywords software verification, software testing, symbolic model checking, abstract reachability, interprocedural analysis, directed search

Contents

Contents i
List of Tables iii
List of Figures v

Chapter 1. Introduction **1**
1.1 Thesis Statement 1
1.2 Approach 2
1.3 Outline 3

Chapter 2. Related Work **4**
2.1 Directed Search 4
2.2 Speed Up ARMC 5
2.2.1 Techniques applicable for speeding up ARMC 5
2.3 Abstract Domain 5
2.4 Recent Verification Techniques other than ARMC 6

Chapter 3. Effective Error Location Directed Search via an Interprocedural Runtime Distance Calculation **7**
3.1 Background 7
3.1.1 Programs 7
3.1.2 Abstract States 7
3.1.3 An ARMC algorithm 9
3.1.4 Function Call Context 9
3.2 Approach 11
3.2.1 The **exit-dist** and **abs-dist** annotation 11
3.2.2 Distance calculation using function call context tracking 19

Chapter 4. Program-specific Distance Metric Selection **22**
4.1 Distance Metrics 22
4.2 Generating a Model for Program-specific Distance Metric Selection 23
4.3 Distance Metric Selection 26

Chapter 5. Empirical Evaluation **27**
5.1 Experimental Setup 27

5.1.1 Research Questions 27
5.1.2 Techniques to Compare 27
5.1.3 Target programs 28
5.1.4 Measurement 29
5.1.5 Model Checking Environmental Setup 30
5.1.6 Threats to validity 30
5.1.7 Limitations 30
5.2 Experimental Result 31
5.2.1 RQ1: Effectiveness of TOUR on bug detection ability . 31
5.2.2 RQ2: Effectiveness of TOUR on verification speed . . . 32
5.2.3 RQ3: Effectiveness of the proposed distance metrics . . 35
5.2.4 RQ4: Effectiveness of the distance metric selection . . . 37
5.2.5 RQ5: Effectiveness of the historical result filtering . . . 37
5.3 Discussion 38
5.3.1 Programs that TOUR is Bad 38
5.3.2 Programs that ARMC cannot Solve 41

Chapter 6. Conclusion **44**
6.1 Future Research Direction 44

Chapter 7. Appendix **45**
7.1 An Effective Implementation of the Directed ARMC of TOUR for Block-Abstraction Memoization 45
7.2 Calculating Program Features 48
7.3 The Extended Program Annotation Algorithms 53

Bibliography **56**

Acknowledgments in Korean **62**

Curriculum Vitae in Korean **63**

List of Tables

4.1 Category and description of the 25 program features used for distance metric selection rule generation; Cyclomatic Complexity: the well-known McCabe's cyclomatic complexity [62] 25

5.1 The statistics of RealWorld.{1,2,3,4}; SMCC: the sum of Cyclomatic Complexity per function 29

5.2 The number of solved buggy RealWorld programs and the total CPU time of BAM and BAM.T; Count.Solved: the number of correctly solved programs; Count.Time-out: the number of programs that a technique fails due to time-out; Count.Others: the number of programs that a technique fails due to reasons other than time-out; Total CPU time: the total model checking time for 354 buggy RealWorld programs 31

5.3 The number of solved buggy RealWorld programs and the total CPU time of LA and LA.T; Count.Solved: the number of correctly solved programs; Count.Time-out: the number of programs that a technique fails due to time-out; Count.Error(recursion): the number of programs that a technique fails due to the recursive function calls; Count.Others: the number of programs that a technique fails due to reasons other than time-out and recursion; Total CPU time: the total model checking time for 354 buggy RealWorld programs 31

5.4 The number of solved clean RealWorld.4 programs and the total CPU time of BAM and BAM.T; Count.Solved: the number of correctly solved programs; Count.Time-out: the number of programs that a technique fails due to time-out; Count.Others: the number of programs that a technique fails due to reasons other than time-out; Total CPU-time: the total model checking time for the 652 clean RealWorld.4 programs 33

5.5 The number of solved programs and total CPU time of each technique for 2,688 clean RealWorld programs; C.SOLV: the number of programs solved by each technique; T.TIME: the total CPU-time of each technique 34

5.6 The number of solved programs and total CPU time of BAM.T for clean RealWorld.{1,2,3,4} programs; C.SOLV: the number of programs solved by BAM.T for each PROG.SET (difference ratio compared to that of BAM); T.TIME: the total CPU-time of BAM.T for each PROG.SET (difference ratio compared to that of BAM) 35

5.7 The number of solved programs and total CPU-time of LA.T for clean RealWorld.{1,2,3,4} programs; C.SOLV: the number of programs solved by LA.T for each PROG.SET (difference ratio compared to that of LA); T.TIME: the total CPU-time of LA.T for each PROG.SET (difference ratio compared to that of LA) 36

5.8 The number of solved programs and total CPU time of BAM.T.{st,bb,lh,lf}; C.SOLV: the number of programs solved; T.TIME: The total CPU time for 3,042 (both buggy and clean) RealWorld programs 36

5.9 The number of solved programs and total CPU time of LA.T.{st,bb,lh,lf}; C.SOLV: the number of programs solved; T.TIME: The total CPU time for 3,042 (both buggy and clean) RealWorld programs 37

5.10 The number of solved programs and total CPU time of BAM.T and BAM.T.{st,bb,lh,lf}; C.SOLV: the number of programs solved; T.TIME: The total CPU time for 3,042 (both buggy and clean) RealWorld programs 37
5.11 The number of solved programs and total CPU time of LA.T.{st,bb,lh,lf}; C.SOLV: the number of programs solved; T.TIME: The total CPU time for 3,042 (both buggy and clean) RealWorld programs 38
5.12 The number of solved programs and total CPU time of BAM.T.{10,20,30,40,50}; C.SOLV: the number of programs solved by BAM.T.{10,20,30,40,50} (difference ratio compared to that of BAM.T.NO that solves 2,081 programs); T.TIME: The total CPU time of BAM.T.{10,20,30,40,50} for 3,042 RealWorld programs (difference ratio compared to that of BAM.T.NO that takes 737,983 seconds of CPU time); N.P: the number of historical results used for the selection model generation 38
5.13 The number of solved programs and total CPU time of LA.T.{10,20,30,40,50}; C.SOLV: the number of programs solved by LA.T.{10,20,30,40,50} (difference ratio compared to that of LA.T.NO that solves 1,481 programs); T.TIME: The total CPU time of LA.T.{10,20,30,40,50} for 3,042 RealWorld programs (difference ratio compared to that of LA.T.NO that takes 729,926 seconds of CPU time); N.P: the number of programs used for the selection model generation 39
5.14 The number of programs in SOLV or NOT-SOLV for each project of RealWorld 41

List of Figures

3.1 An example C program . . . 8
3.2 An example abstract state graph for the example program in Figure 3.1 . . . 8
3.3 An example abstract reachability-based model checking algorithm . . . 10
3.4 Overview of TOUR (effectTive error lOcation directed search via an interprocedUral Run-time distance calculation) . . . 11
3.5 The control-flow graph of the example program in Figure 3.1 labeled with exit-dist and exit-dist′ . . . 12
3.6 The control-flow graph of the example program in Figure 3.1 labeled with abs-dist . . . 13
3.7 Algorithm: `exit-dist` . . . 14
3.8 An example order of the exit-dist annotation of functions, i.e., x, y, and main, using a function call dependency graph . . . 15
3.9 Algorithm: `abs-dist` . . . 16
3.10 An example order of calling `exit-dist` and `exit-dist*` using function call dependency graph to handle recursive function calls . . . 17
3.11 Algorithm: `exit-dist*` . . . 18
3.12 An example abstract state graph with tracked function call context and calculated distance 20
3.13 The directed ARMC algorithm . . . 21

4.1 Overview of TOUR with the program-specific distance metric selection . . . 22
4.2 The procedure for generating a model for program-specific distance metric selection from historical results . . . 24
4.3 Preprocessing the historical results of directed model checking; TO, time-out; the shaded row, filtered out by min-max threshold 10 seconds . . . 26
4.4 An example distance metric selection model represented as a decision tree . . . 26

5.1 The scatter plot of a baseline technique's CPU time (x-axis) and a TOUR technique's CPU time (y-axis) in log-scale (a) for 97 RealWorld buggy programs that both BAM.T and BAM find bugs; (b) for 42 RealWorld buggy programs that both LA.T and LA find bugs . . . 32
5.2 The CPU time of BAM (x-axis) and BAM.T (y-axis) in log-scale for 323 clean RealWorld.4 programs that both BAM and BAM.T verify . . . 33
5.3 The scatter plot of a baseline technique's CPU time (x-axis) and a TOUR technique's CPU time (y-axis) in log-scale (a) for 1900 clean RealWorld programs that both BAM.T and BAM verify; (b) for 1398 clean RealWorld programs that both LA.T and LA verify . 35
5.4 (Left) For 1,825 RealWorld programs that both BAM and BAM.T.st solve, the CPU time difference between BAM and BAM.T.st in log scale for y-axis and the constructed states difference between BAM and BAM.T.st in log scale for x-axis; (Right) For 1,978 RealWorld programs that both BAM and BAM.T.lf solve, the CPU time difference between BAM and BAM.T.lf in log scale for y-axis and the constructed states difference between BAM and BAM.T.lf in log scale for x-axis . . . 40

5.5 (Left) For 1,825 RealWorld programs that both BAM and BAM.T.st solve, the number of coverage-check difference between BAM and BAM.T.st in log scale for y-axis and the constructed states difference between BAM and BAM.T.st in log scale for x-axis; (Right) For 1,978 RealWorld programs that both BAM and BAM.T.lf solve, the number of coverage-check difference between BAM and BAM.T.lf in log scale for y-axis and the constructed states difference between BAM and BAM.T.lf in log scale for x-axis 40
5.6 (a) The AVNDFN value distribution of NOT-SOLV programs in each project of RealWorld; (b) The NODES value distribution of NOT-SOLV programs in each project of RealWorld 42
5.7 The NODES, SMCC, VARSINC, VARSLOOP, and FIELDS value distribution of NOT-SOLV and SOLV programs in LDV of RealWorld 42
5.8 The NODES value distribution of NOT-SOLV and SOLV programs in AWS of RealWorld 43

7.1 An example C program 46
7.2 An example abstract state graph constructed by BAM for the example program in Figure **??**, which shows the problem of applying the directed ARMC of TOUR to BAM . . . 47
7.3 The directed ARMC algorithm of TOUR extended with the mapping function `entry` . . . 49
7.4 Algorithm: `dist` 49
7.5 An example C program 52
7.6 The control-flow graph of the example program in Figure 7.5 53
7.7 Algorithm: `exit-dist` 54
7.8 Algorithm: `abs-dist` 55

Chapter 1. Introduction

Model checking is an effective approach to verify whether a program reaches an error location (i.e., reachability), but it often suffers from the state-explosion problem [1, 2, 3]. Model checking evaluates all program inputs to enumerate all reachable program states to verify a reachability property. Since a real-world program has huge or even infinite reachable states, a model checking technique often fails to enumerate all reachable program states due to exceeded resource limits (i.e., the state-explosion problem).

Abstract Reachability-based Model Checking (ARMC) techniques conduct the reachability analysis on an abstraction of an actual program state space, i.e., a finite and relatively small abstract state space, to mitigate the state-explosion problem [4, 5, 6, 7, 8, 9, 10, 11, 12, 13]. Although many studies reduce the ARMC model checking time [9, 15, 16, 17, 18], time-outs are the reason for 75% (772 out of 1,031) of cases where the state-of-the-art ARMC technique (i.e., block-abstraction memoization for interprocedural analysis [14]), combining their benefits, fails to verify in our preliminary experiment on real-world C programs.

Existing ARMC techniques are slow to detect a path reaching an error location (i.e., a target path) due to an inefficient search method. A target path is either an actual bug path (i.e., a counterexample of the reachability property) or a false alarm triggering the refinement of the abstraction [19, 20]. If a target path is an actual bug path, ARMC finishes with the result where the program can reach an error location and the target path is a counterexample of the property violation. If a target path is false alarm, ARMC builds another refined abstraction that does not have the detected false alarm. ARMC refines the abstraction until it finds a bug path or an abstraction has no false alarm. Thus, I can improve ARMC in terms of both bug detection ability and verification speed by detecting a target path quickly. However, existing ARMC techniques explore an abstract state space with an inefficient search method, e.g., depth-first search, for detecting a target path [21, 22].

Nevertheless, to the best of our knowledge, no previous work studies about the fast target path detection in real-world software. For low-level state transition systems, some techniques utilize a target directed search for better bug finding ability, which is not applicable to high-level C programs [23, 24, 25]. Some symbolic execution techniques for C program utilize a target directed search, but their search methods does not fully applicable to ARMC techniques due to the abstraction [26, 27]. A study applies a target directed search to an ARMC technique, but it does not support interprocedural analysis although most of real-world C programs have more than one procedures [28].

1.1 Thesis Statement

The goal of this research is to improve both bug detection ability and verification speed of ARMC techniques for real-world programs by detecting a target path quickly using an effective search method. The key idea is an error location directed search that utilizes the distance to an error location and function call context at runtime. An error location directed search is a candidate solution for fast target path detection. It tries to find a target path with the shortest syntactic distance first. It reduces the number of explored states irrelevant to a target path although a path with the shortest syntactic distance may be an invalid path in program semantics. An error location directed search should consider function call contexts because most of real-world C program are an interprocedural program and have various function call contexts. This leads to the following thesis statement to achieve the goal:

An *error location directed search* using *the distance to an error location* and *function call context* at runtime improves both bug detection ability and verification speed of ARMC techniques.

1.2 Approach

To investigate the thesis, this research follows three steps.

First, I propose a novel technique TOUR (effecTive error lOcation directed search via an inter-procedUral Runtime distance calculation), which is the first approach to apply a target directed search for fast target path detection in ARMC for real-world programs. TOUR explores an abstract search space by first selecting a state with the shortest distance to an error location. Note that TOUR calculates the distance by considering the function call context at runtime because an execution path of an interprocedural program consists of various function call contexts.

TOUR calculates distances of two different types – one is dependent on the function call context (rel-dist) and the other is independent from the context (abs-dist): ① the rel-dist value is the sum of the distance from the function call abstract state to the error location and the distance from the abstract state to the function exit location; and ② the abs-dist value is the distance from the abstract state to the error location without considering the function call context. TOUR labels each program location with abs-dist and exit-dist where exit-dist is the distance from a location to the function exit location. The distance of an abstract state is the smaller of rel-dist and abs-dist. TOUR tracks function call context (i.e., function call abstract states) for each abstract state at runtime for rel-dist calculation. TOUR caches the distance of function call abstract states for efficient rel-dist calculation.

Second, to further improve the directed search strategy of TOUR, I propose four different distance metrics and a program-specific distance metric selection method by learning from historical results of TOUR. The four distance metrics are number of statements, number of basic blocks, number of loop heads, and number of loop heads and function entry/exit points. Each metric has an advantage for different types of ARMC techniques and programs. Note that a metric may not be the best for a program even if the metric is generally good for an ARMC technique. Thus, TOUR generates a program-specific distance metric selection heuristic by learning from historical results with 25 static program features. Note that this is the first research that compares various directed search heuristics for ARMC.

Third, I extensively evaluate TOUR on 3,042 real-world C programs from a software verification competition (i.e., SV-COMP 2021) to show the improved bug detection ability and verification speed. I implemented TOUR on top of two existing ARMC techniques, Block-Abstraction Memoization for interprocedural program analysis (BAM) and Lazy Abstraction (LA). The 3,042 C programs are from SoftwareSystems category of the SV-COMP 2021 benchmark which contains real-world verification tasks. The experimental results show that BAM using TOUR finds bugs in 20% more programs within 11% less total CPU-time for 354 real-world buggy C programs, which shows the improved bug detection ability. BAM using TOUR verifies 4% more program within 12% less total CPU-time for 2,688 real-world clean C programs, which shows the improved verification speed. For 652 complex real-world clean C programs classified by the Cyclomatic Complexity, BAM using TOUR verifies 15% more programs within 15% less total CPU-time, which shows increasing effectiveness as increased complexity of programs. BAM using TOUR with the distance metric selection solves 10% more programs within 22% less total CPU-time for

3,042 real-world C programs compared to the case using the worst single distance metric.

1.3 Outline

The rest of this dissertation is organized as follows. Chapter 2 introduces related work. Chapter 3 describes the details of the error location directed search of TOUR. Chapter 4 explains the distance metrics that TOUR uses and the program-specific distance metric selection. Chapter 5 demonstrates the empirical evaluation process and results of TOUR. Chapter 6 concludes the research with future research direction.

Chapter 2. Related Work

In this chapter, I introduce studies related to directed search methods, speed-up methods for ARMC, and abstract domains.

2.1 Directed Search

Although several program analysis techniques (e.g., model checking with distance-preserving abstractions, symbolic executions, or directed search methods for intraprocedural programs) utilize directed search strategies, TOUR is the *first ARMC technique for real-world programs* that applies directed search for fast target path detection on real-world programs.

Directed model checking with distance-preserving abstractions utilizes directed search, but it targets a state transition system, not a high-level C program [23]. It abstracts a state transition system by grouping states with the same distance to an error state to mitigate the state-explosion problem. TOUR is naturally distance-preserving because TOUR does not abstract the program counter variable of a target C program (i.e., the program location) for which TOUR defines the distance. The distance-preserving abstraction technique uses only a single distance metric, while TOUR uses and compares four different distance metrics.

Shortest-Distance Symbolic Execution (SDSE) applies directed search to symbolic execution, while TOUR applies a directed search to ARMC [27]. SDSE computes the distance to an error location in an interprocedural C program and conducts the shortest error distance first search. SDSE applies a directed search for fast bug path detection, but TOUR improves both bug detection ability and verification speed for ARMC. SDSE also uses a single distance metric (i.e., the number of edges), while TOUR uses four distance metrics.

Call Chain Backward Symbolic Execution (CCBSE) is another directed symbolic execution technique, but CCBSE is hard to apply to ARMC [27]. CCBSE first explores a partial path from the start of a function `f` including an error location to the error location. CCBSE then expands the partial path in backward for each function `g` that invokes `f` to find a feasible path from the program start to the error location. CCBSE however utilizes the backward state search, not yet supported in ARMC. CCBSE also uses a single distance metric, i.e., the number of edges.

A study applies an error location directed search for ARMC on intraprocedural programs while TOUR targets ARMC for interprocedural programs [28]. It first explore an abstract state with the shortest distance to the error location. However, it only calculate the distance on a intraprocedural program while most of the real-world programs are interprocedural. It also uses a single distance metric, i.e., the number of edges, for the distance calculation.

UFO is an ARMC technique that uses a search strategy for fast loop unwinding [29]. Its search strategy is called recursive iteration strategy [30]. It always iterates a innermost loop first to summarize the loop quickly. The recursive iteration strategy is effective for UFO because UFO handles a chunk of program statements without a cycle at once. UFO utilizes state-joining to reduce the number of program paths [31].

2.2 Speed Up ARMC

Although there are several techniques to speed up ARMC (e.g., lazy abstraction, block-encoding, and block-abstraction memoization), TOUR improves the ARMC's speed further as a *complementary* optimization technique.

TOUR further improves the speed of Block-Abstraction Memoization for interprocedural program analysis (BAM), even though BAM is the state-of-the-art ARMC technique combining the benefits of existing optimization techniques [14]. BAM combines three existing speed-up techniques for ARMC, i.e., Lazy Abstraction (LA), Block-Encoding (BE), and block-abstraction memoization. LA reuses the abstract states that are not relevant to a false alarm after removing the false alarm to save time for state exploration [9, 11]. BE takes advantage of the efficiency of constraint solving techniques: it is more efficient to compute an abstract state for a large block of multiple statements at once than to compute abstract states of the multiple statements separately with multiple computations [13, 12]. Block-abstraction memoization stores constructed substate graphs to a cache and reuses them later to save state exploration time [40]. TOUR complements the three speed-up techniques because it tries to search as few states as possible instead of reusing or efficiently constructing them.

2.2.1 Techniques applicable for speeding up ARMC

Some studies are not designed for speeding up ARMC, but they are also applicable to ARMC and may be effective for speeding up ARMC.

Refinement selection diversifies information to be used for refinement to avoid conducting an inefficient refinement [32, 33]. A refinement is inefficient if a spurious counterexample similar to the removed spurious counterexample by the refinement occurs again later. Such inefficient refinements occur because existing refinement techniques eliminate spurious counterexamples using a variable that is not directly related to the verification property, such as a loop counter variable. The technique breaks down the reason for the spurious error state and selects the most relevant variables for the verification property to address the above problem.

Beautiful interpolants proposed a method producing a more sophisticated form of abstract data states in refinements, such as $a + b < 3$, whereas an existing ARMC techniques produces only a simple form of abstract data states like $i < 1$ [34]. By creating this complex form of abstract data states, the technique aims to replace multiple refinements with one more efficient refinement.

2.3 Abstract Domain

TOUR is *independent* of abstract domains and thus can be applied to ARMC techniques with various abstract domains.

I successfully apply TOUR to BAM, which uses two different abstract domains (i.e., predicate domain and explicit-value domain) [14] because TOUR is independent of abstract domains. The predicate domain is a widely used abstract domain for ARMC of C programs [35, 6, 36, 11, 5, 12, 37, 17]. The predicate domain presents an abstract data state formula as a set of predicates over program variables. Most of the verifiers using predicate domain utilize Craig interpolation for abstract data state generation [41]. It is also effective for unbounded loop analysis because it easily checks whether a loop fixpoint is reached [6, 38, 39]. The explicit-value domain uses an explicit-value assignment of some program variables (not all the program variables) as an abstract data state formula [15, 33, 16]. It is effective to verify a program in which the reachability of the program is primarily related to some specific variables. TOUR is independent of the domains because the domains affect only the abstract data state formula and maintain program locations as nonabstracted. I also apply TOUR to LA which uses the predicate abstract domain [9].

Although TOUR can be applied to LAI, it may not be effective to apply TOUR to Lazy Abstraction with Interpolants (LAI). LAI is an ARMC technique that uses the empty abstract domain for the entire model checking procedure [42, 43, 44]. LAI cannot compute abstract data states during state exploration because of the empty abstract domain. It computes abstract data states during the removal of false alarms. TOUR saves time during state exploration, which is not effective for LAI because LAI does not have much time for exploration due to the empty abstract domain.

Applying TOUR to a combination-based model checking technique (e.g., CPA-seq and PeSCo) is not effective because the technique may assign a short time to an ARMC component constituting the technique. Combination-based techniques assume that a suitable abstract domain for model checking of a program may differ program by program. CPA-seq sequentially invokes model checking techniques, including ARMC components and other model checking techniques, to improve the chance of solving a program [45, 46]. PeSCo [47, 48] extends CPA-seq by dynamically changing the order of constituting techniques by predicting the probability of solving. Both CPA-seq and PeSCo assign a short time limit (only 11% (= 100s/900s) of the total time limit) to an ARMC component because they think that a technique is not suitable for a program if it cannot solve the program quickly. However, TOUR helps ARMC solve a complex program with reasonable time, while ARMC fails to verify the program due to time-out.

MUX is also a verifier selection technique that does not sequentially combine verifiers [51]. It simply selects a verifier for a given target program. It uses a machine learning technique and learns from historical model checking results of various verifiers and programs. It targets windows device driver programs.

Conditional model checking and reducer-based model checker encourage to utilize multiple verifiers with various abstract domain for a program verification by allowing information exchange between verifiers [49, 50]. They focus on how to deliver an intermediate abstract state graph to other verifiers without abstract domain specific information. They summarize an intermediate abstract state graph as an actual C program used as a usual target program of other verifier.

Dynamic precision adjustment encourages to utilize combined abstract domain [52]. It allows to use multiple abstract domain for a program verification. At each time of generating abstract data state, the technique uses an appropriate abstract domain among the supported domains. Note that it cannot exchange the information between different domains while it manages each domain separately.

2.4 Recent Verification Techniques other than ARMC

SMACK is a framework to modularize verifiers [53]. It verifies a program presented as an intermediate language and provides various interfaces from existing verification front-ends and back-ends. Thus, SMACK easily accepts existing verification technique's benefits and integrates them into SMACK.

Bounded model checking techniques are still competitive for many verification tasks [56, 57, 58]. They utilize k-induction to increase the bound of verification. When the verification reaches a bound, k-induction summarize the reached state of the bound as a simple abstract state. After that k-induction continues the bounded model checking from the summarized abstract state to reach the bound again.

Chapter 3. Effective Error Location Directed Search via an Interprocedural Runtime Distance Calculation

3.1 Background

This section explains background knowledge for understanding TOUR (effecTive error lOcation directed search via an interprocedUral Runtime distance calculation), including programs, abstract states, and function call context.

3.1.1 Programs

A program consists of *locations* that model program counter variables and *operations* executed when a control moves from a location to another location . Figure 3.1 shows an example C program consisting of `main` and `f`, which is labeled with program location numbers. I assume simple interprocedural C programs with limited operations: assignment operations (e.g., `L5` $\rightarrow$ `L7`), branch condition operations (e.g., `L2` $\rightarrow$ `L3`, `L2` $\rightarrow$ `L8`), function call operations (e.g., `L3` $\rightarrow$ `L21`), function return operations (e.g., `L22` $\rightarrow$ `L4`), and dummy operations (e.g., `L7` $\rightarrow$ `L10`) that move program counters without changing program states. For each operation, there are a *source* location and a *destination* location. For example, for the assignment operation `L5` $\rightarrow$ `L7`, `L5` is the source location and `L7` is the destination location. A location l' is a *successor* location of l (and l is a *predecessor* location of l') if l is the source location of an operation op and l' is the destination location of op. For example, `L3` and `L8` are successor locations of `L2` and `L2` is a predecessor location of them. A location l is a *predecessor* location of l' if l A function has one *function entry location* (e.g., `L1`, `L21`) and one *function exit location* (e.g., `L12`, 22). The location `L11` is the *error location* which unwillingly terminates the program by invoking the `abort` function. The source location of a function call operation is a *function call location* (e.g., `L3` and `L8`) and the destination location of a function return operation is a *function return location* (e.g., `L4` and `L9`).

3.1.2 Abstract States

An *abstract state* consists of a corresponding location and an abstract data state describing reachable program states under an abstraction [12]. Figure 3.2 shows an example abstract state graph for the program in Figure 3.1, abstracting all operations as nondeterministic (i.e., any program state is reachable after executing an operation, represented as the formula $True$). A graph node denotes an abstract state with the corresponding location (left) and the abstract data state formula (right). A graph edge denotes a *successor/predecessor* relationship between two abstract states. It also represents the operation between the location of the successor abstract state and that of the predecessor abstract state. The *initial abstract state* s_1 corresponds to the function entry location of the `main` function (i.e., `L1`) and always has $True$ as the abstract data state formula. A *target path* is the sequence of abstract states from the initial abstract state to an abstract state corresponding to an error location with a non$False$ abstract data state formula (e.g., $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \rightarrow s_8 \rightarrow s_9 \rightarrow s_{10}$).

```
     void main(int *x,int k){
 L1:   int i;
 L2:   if(x[1]>x[2]){
 L3:     f(x[1]);
 L4:     if(k==0){
 L5:       x[2]=x[1];
         }else{
 L6:       x[1]=x[2];
 L7:     }
       }else{
 L8:     f(x[2]);
 L9:   }
L10:   if(x[1]>x[2]){
L11:     abort();
       }
L12:}
     void f(int i){
L21:   int a=i;
L22:}
```

Figure 3.1: An example C program

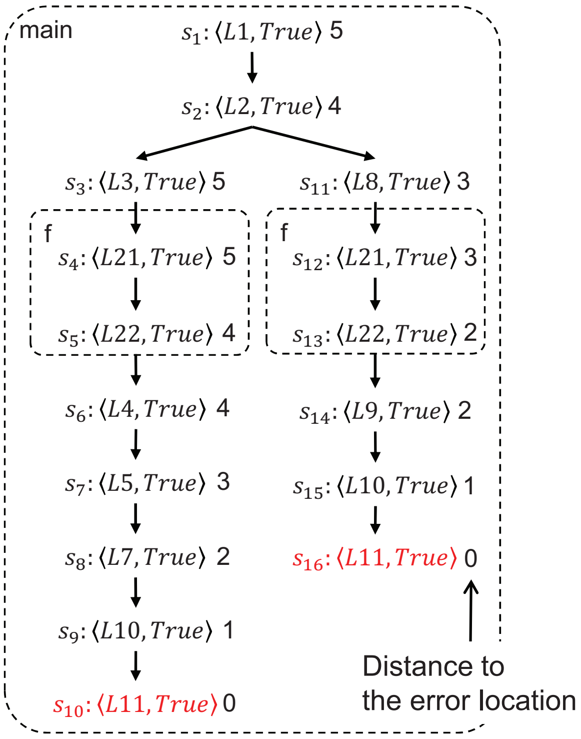


Figure 3.2: An example abstract state graph for the example program in Figure 3.1

3.1.3 An ARMC algorithm

An ARMC algorithm takes an abstract state graph consists of a set of graph nodes N and a set of graph edges E, and a set of error locations L_{err}. It explores abstract state space from the input graph and returns $safe$ if the abstract state space (i.e., the abstract state graph) has no target path or $unsafe$ if the abstract state space has a target path.

Figure 3.3 shows an ARMC algorithm modified and simplified from the algorithm in [59]. The algorithm initializes the set of reached abstract state graph nodes reached and the set of unmarked abstract state graph nodes waitlist (line 1–2). An abstract state is marked if the algorithm finds all the successor states of the abstract state. Thus, the unmarked nodes are considered as the states waiting to be visited for finding successors. The algorithm repeats until waitlist is empty by finding successors of an abstract state selected from waitlist. At the start of each iteration, the algorithm takes an abstract state from waitlist (line 4). Then the algorithm finds the successor abstract states of the selected one by following each outgoing operations of the selected one. The algorithm constructs an abstract state according to the operation (line 6). It adds the newly explored abstract state to the graph by adding the corresponding state graph node and the corresponding graph edge (line 7–8). If the abstract data state of the newly explored abstract state is not $False$ (i.e., the abstract state is reachable) and the abstract state is corresponding an error location, the algorithm returns $\langle unsafe, \mathsf{reached}, E\rangle$ because it is the detected target path (line 11). If the abstract state does not correspond to an error location, the algorithm put the state to waitlist for further state space exploration (line 13). If the abstract data state is $False$, the constructed abstract state is not reachable and the algorithm skips to put the abstract state to waitlist (line 9). After finding all the successors of the selected abstract state, the algorithm mark the selected abstract state (line 16). If there are no more abstract states in waitlist, the algorithm finishes with $safe$ because there is no reachable error location and no more expandable abstract state (line 18).

Note that it is necessary to handle a detected target path for sound reachability analysis because a target path on an abstract state space might not be reproducible in actual program due to the abstraction [20]. However, that part is out-of-scope of this study.

3.1.4 Function Call Context

A *function call context* of an abstract state s is the function call abstract state of s denoted as `call`(s). For example in Figure 3.2, the function call context of s_4 and s_5 is s_3 because they correspond to the function f invoked at s_3.

Abstract states corresponding to the same location can have different distance values on different function call contexts [27]. For example in Figure 3.2, I annotate the distance from each abstract state to the error location in terms of the number of statements (i.e., the number of control-flow edges). The abstract states s_4 and s_{12} have different distance values due to the different function call contexts, although they correspond to the same location L21.

An error location directed search considering the function call context can find a target path faster than other search methods. For example, in Figure 3.2, the shortest distance first directed search finds the target path from s_1 to s_{16} by calculating the distance on function call contexts. It is shorter than the target path from s_1 to s_{10} detected by the depth-first search with a true branch first manner.

```
Input: a set of abstract state graph nodes $N$; a set of abstract state graph edges $E$; a set of error
    locations $L_{err}$
Output: $safe$ if the graph contains no target path or $\langle unsafe, N, E\rangle$ if the graph contains a target path
 1: reached := $N$
 2: waitlist := unmarked nodes from $N$
 3: while waitlist$\neq \emptyset$ do
 4:     $\langle l, a\rangle$ := pop an abstract state from waitlist
 5:     for all outgoing operations $op$ of $l$ do
 6:         $\langle l', a'\rangle := \langle l, a\rangle$'s successor abstract state according to $op$
 7:         reached := reached $\cup \{\langle l', a'\rangle\}$
 8:         $E := E \cup \{\langle l, a\rangle \rightarrow \langle l', a'\rangle\}$
 9:         if $a' \neq False$ then
10:             if $l' \in L_{err}$ then
11:                 return $\langle unsafe,$ reached$, E\rangle$
12:             end if
13:             waitlist := waitlist $\cup \{\langle l', a'\rangle\}$
14:         end if
15:     end for
16:     mark $\langle l, a\rangle$
17: end while
18: return $safe$
```

Figure 3.3: An example abstract reachability-based model checking algorithm

3.2 Approach

Figure 3.4 shows the graphical overview of TOUR. TOUR calculates the two types of distance values, i.e., rel-dist and abs-dist, at runtime. TOUR annotates information necessary for the calculation to a target program before runtime. The annotated information is the abs-dist of a program location to an error location and the distance from a location to the function exit location (exit-dist). TOUR tracks the function call context, i.e., which abstract state is a function call abstract state of which abstract states, during the run-time for the rel-dist calculation.

TOUR calculates the two distance values (rel-dist and abs-dist) that complement each other. An abstract state has no abs-dist if its distance is only valid under a function call context. The rel-dist value of an abstract state is not the shortest distance if the abstract state has a valid abs-dist value smaller than the rel-dist value. TOUR selects the smaller distance of rel-dist and abs-dist as the abstract state distance.

The abs-dist value of an abstract state is the abs-dist value of the corresponding program location. The rel-dist value of an abstract state is the sum of the exit-dist and the distance of the corresponding function call of the abstract state. Each abstract state tracks its corresponding function call to the abstract state at runtime for rel-dist calculation. TOUR annotates the information (i.e., exit-dist and abs-dist) necessary for calculating the distance to a target program for runtime calculation efficiency.

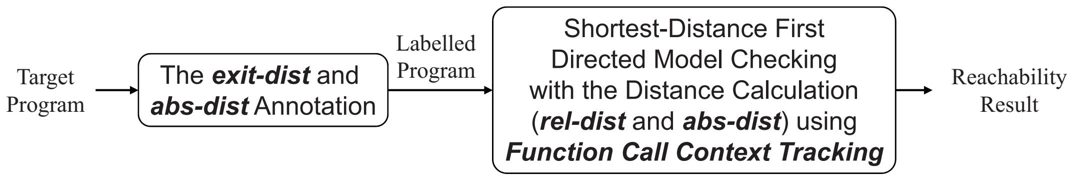

Target Program
The exit-dist and abs-dist Annotation
Labelled Program
Shortest-Distance First Directed Model Checking with the Distance Calculation (rel-dist and abs-dist) using Function Call Context Tracking
Reachability Result

Figure 3.4: Overview of TOUR (effectTive error lOcation directed search via an interprocedUral Runtime distance calculation)

3.2.1 The exit-dist and abs-dist annotation

The exit-dist value of a program location is the shortest distance from the location to the function exit location according to the selected distance metric. TOUR calculates and annotates exit-dist for each function of a target program by using a single-source shortest distance calculation algorithm in the backward direction. The function exit location is the single source of the algorithm.

Figure 3.5 shows an example exit-dist annotation for the example program in Figure 3.1. It describes the control-flow graph of the example program labeled with exit-dist, where a node represents a program location. TOUR applies the exit-dist annotation algorithm for each function f and main in order because f has no callee function and is the callee function of main. All the edges have same weight except the function call and return edges that have zero weight (i.e., L3 → L21, L8 → L21, L22 → L4, and L22 → L9). TOUR initializes the exit-dist value of the function exit location as zero and that of other locations as ∞. The single source of the algorithm for the function f is L22 and that for the function main is L12. The exit-dist value of a location l is the sum of the exit-dist of l's successor location and the edge weight between them. The algorithm ignores function return edges (e.g., L22 → L4 and L22 → L9). TOUR calculates the exit-dist value of a function call location (e.g., L3 and L8) as the sum of the exit-dist value of the corresponding function return location (e.g., L4 and L9 respectively) and the function's

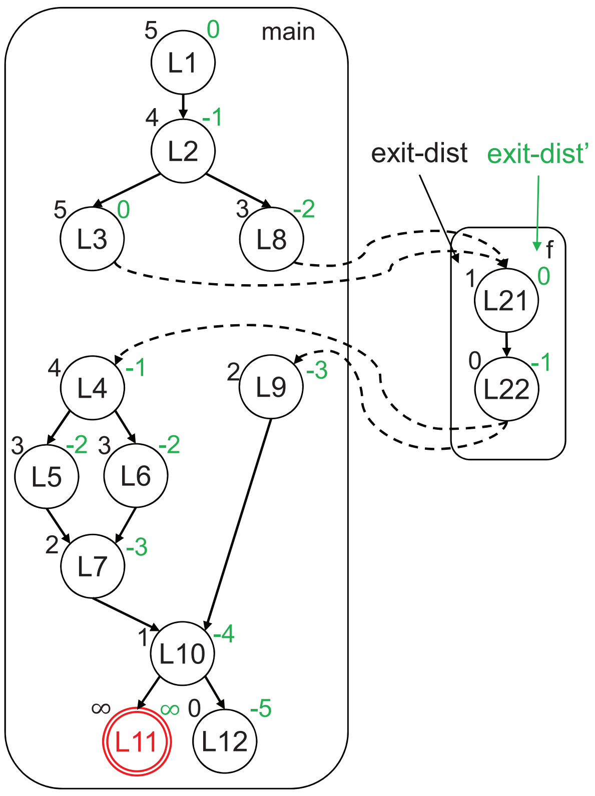


Figure 3.5: The control-flow graph of the example program in Figure 3.1 labeled with exit-dist and exit-dist'

shortest distance (e.g., the function f). A function's shortest distance means the shortest distance from the function entry to the function exit location (i.e., the exit-dist of the function entry location, e.g., L21). Thus, TOUR calculates the exit-dist for a callee function first.

After the exit-dist annotation, TOUR calculate exit-dist' for the runtime distance calculation. The exit-dist' of a location l is l's exit-dist minus the exit-dist of the function entry location. For example, in Figure 3.5, L22's exit-dist' is -1 $(= 0 - 1)$ that is the exit-dist of L22 minus L21's exit-dist (i.e., the function entry location of the function f that includes L21 and L22). The exit-dist' of a location l is used to calculate the distance of an abstract corresponding to l (i.e., $\langle l, a\rangle$ where a is an abstract data state formula). For example, in Figure 3.5, if the function main calls f with the distance 5, the distance of L22 is 4 $(= 5 - 1)$ that is the function call distance minus the exit-dist' of L22.

The abs-dist value of a program location is the shortest distance from the location to an error location according to the selected distance metric. TOUR calculates and annotates abs-dist to a target program using a single source shortest distance algorithm in the backward direction. An error location is the single source of the algorithm.

Figure 3.6 shows an example abs-dist annotation for the example program in Figure 3.1. It describes the control-flow graph of the example program labeled with abs-dist. TOUR initializes the abs-dist value of the error location (L11) as zero (because it is the single source of the algorithm) and that of other locations as ∞. The abs-dist value of a location l is the sum of the abs-dist of l's successor location and the edge weight between them. The algorithm ignores the function return edges, L22 $\rightarrow$ L4 and L22 $\rightarrow$ L9. TOUR calculates the abs-dist value of a function call location (e.g., L3 and L8) as the sum of the abs-dist value of the corresponding function return location (e.g., L4 and L9, respectively) and

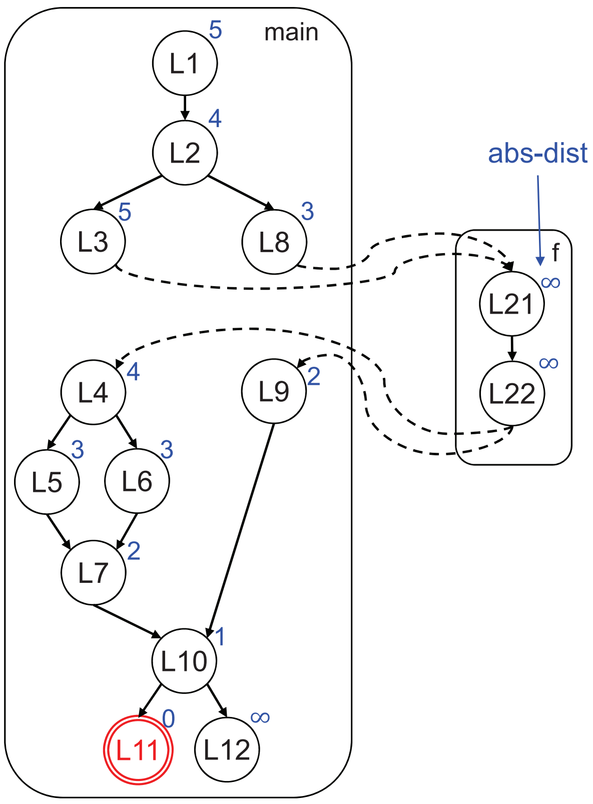


Figure 3.6: The control-flow graph of the example program in Figure 3.1 labeled with abs-dist

the function's shortest distance (e.g., the function f). Since the abs-dist calculation algorithm uses the exit-dist values (as the function's shortest distance), TOUR calculates exit-dist for all functions first and then calculates abs-dist.

The Program Annotation Algorithms

Figure 3.7 describes the exit-dist annotation algorithm for a given function f. The algorithm takes f, the function name to compute exit-dist and uses global variables; and FW, the mapping function from function name to the function's shortest distance (i.e., exit-dist of the function entry location).

The algorithm is basically a single-source shortest-path algorithm for a directed graph with non-negative edge weights. The function exit location l_{exit} is the single-source of the algorithm (i.e., l_{exit}.exit-dist$= 0$) because exit-dist is the distance from a location to the function exit location (line 3). The set VS stores visited locations to avoid duplicate exit-dist annotation of a location. The set Q stores locations to be visited, i.e., locations of which the algorithm does not annotate the exit-dist of the source locations.

The algorithm repeatedly visits locations and annotates exit-dist to each location until Q is empty (line 8). At the beginning of each iteration, the algorithm takes a location l with the smallest exit-dist from Q (line 9). For each incoming operation e of l, the algorithm identify the source location p of e, i.e., a predecessor location of l (line 11). If p is already visited (line 12–14), the algorithm skips the operation e to avoid the duplicate annotation. The algorithm annotates the exit-dist of p with regard to the type of the operation e. If e is not a function return operation, the exit-dist of p is the exit-dist of l plus one (line 16). If e is a function return operation (i.e., l is a function return location), the algorithm reassign p as the l's corresponding function call location (line 18). In that case, the exit-dist of p (i.e.,

```
Input: $f$, the function name to compute exit-dist; Global Variables: $FW$, the mapping function from
    function name to the function's shortest distance
1: $l_{exit}$ := the function exit location of $f$
2: $l_{entry}$ := the function entry location of $f$
3: $l_{exit}$.exit-dist$:= 0$
4: $VS := \emptyset$
5: $Q := \emptyset$
6: $VS := VS \cup \{l_{exit}\}$
7: $Q := Q \cup \{l_{exit}\}$
8: while $Q \neq \emptyset$ do
9:    $l$ := pick a location with the smallest exit-dist from $Q$
10:   for all incoming operations $e$ of $l$ do
11:      $p := e$'s source location
12:      if $p \in VS$) then
13:         continue
14:      end if
15:      if $e$ is not a function return operation then
16:         $p$.exit-dist$:= 1 + l$.exit-dist
17:      else
18:         $p$ := the corresponding function call location of $l$
19:         $p$.exit-dist$:= FW(p'$s function name$) + l$.exit-dist
20:      end if
21:      $VS := VS \cup \{p\}$
22:      if $p \neq l_{entry}$ then
23:         $Q := Q \cup \{p\}$
24:      end if
25:   end for
26: end while
27: $FW(f) := l_{entry}$.exit-dist
```

Figure 3.7: Algorithm: `exit-dist`

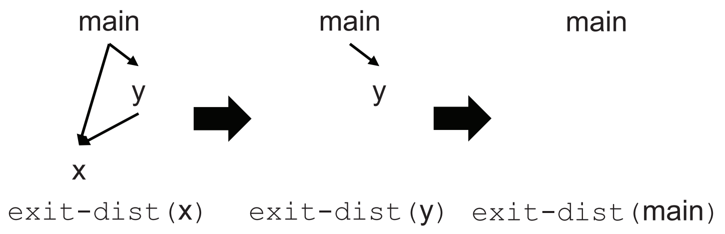


Figure 3.8: An example order of the exit-dist annotation of functions, i.e., x, y, and main, using a function call dependency graph

the l's corresponding function call location) is the sum of the shortest distance of the p's function (i.e., $FW(p'\text{s function name})$) and the l's exit-dist (line 19). After annotating p's exit-dist, the algorithm put p to VS (line 21). If p is not the function entry location of f, the algorithm put p to Q to annotate exit-dist to the successor locations of p (line 23). The algorithm does not put the function entry location to Q to annotate exit-dist within a function.

If Q is empty, the algorithm finishes because it labels all the locations of f with exit-dist. The algorithm assign $FW(f)$ as l_{entry}'s exit-dist for the exit-dist calculation of f's caller functions.

Note that the algorithm `exit-dist` is sound only if FW has all the callee functions' shortest distance of f. Thus, the exit-dist annotation of functions should be ordered according to the function call dependency graph. Figure 3.8 shows an example function call dependency graph and the order of the `exit-dist` algorithm calls. At first, x is the only function that has no callee function among x, y, and main. After the `exit-dist`(x) call, x is removed from the dependency graph because FW has the shortest distance of the function x. At that time, the function y has no callee function so that `exit-dist`(y) becomes available. The function y is removed after the `exit-dist`(y) call and finally `exit-dist`(main) becomes available.

Figure 3.9 describes the abs-dist annotation algorithm for an interprocedural program. It takes the set of error locations L_{err} and the mapping function from function name to the function's shortest distance FW. The algorithm should be invoked after completing the exit-dist annotation of all functions in the program because the algorithm needs FW of all the functions.

The algorithm is basically a single-source shortest-path algorithm for a directed graph with non-negative edge weights. VS stores visited locations to avoid duplicate abs-dist annotations (line 1). Q stores locations of which the algorithm not yet annotates the abs-dist of the predecessor locations (line 2). An error location can be the single source of the algorithm because abs-dist means the distance from a location to the error location. Suppose there is more than one error location, the algorithm assumes a dummy single-source location considered as the successor location of all error locations connected with dummy operations. Thus, the algorithm put each l_{err} in L_{err} to VS and Q with the zero abs-dist (line 3–7).

The algorithm repeatedly visits locations and annotates abs-dist Q is empty (line 8). At the beginning of each iteration, the algorithm selects a location with the smallest abs-dist from Q (line 9). For all incoming operations of the selected location l, the algorithm finds the predecessor location p of l to annotate the abs-dist to p (line 11). If p is in VS, the algorithm skips the abs-dist annotation of p to avoid the duplicate annotation (line 13). If e is not a function return operation, the algorithm calculates p's abs-dist as the l's abs-dist plus one (line 16). If e is a function return operation (i.e., l is a function return location), the algorithm reassign p as the l's corresponding function call location (line 18). And then the algorithm calculates p's abs-dist as the sum of the shortest distance of the p's function (i.e.,

```
Input: $L_{err}$, the set of error locations; Global Variables: $FW$, the mapping function from function
    name to the function's shortest distance;
1: $VS := \emptyset$
2: $Q := \emptyset$
3: for all $l_{err} \in L_{err}$ do
4:   $l_{err}$.abs-dist$:= 0$
5:   $VS := VS \cup \{l_{err}\}$
6:   $Q := Q \cup \{l_{err}\}$
7: end for
8: while $Q \neq \emptyset$ do
9:   $l :=$ pick a location with the smallest abs-dist from $Q$
10:  for all incoming operations $e$ of $l$ do
11:    $p := e$'s source location
12:    if $p \in VS$ then
13:      continue
14:    end if
15:    if $e$ is not a function return operation then
16:      $p$.abs-dist$:= 1 + l$.abs-dist
17:    else
18:      $p :=$ the corresponding function call location of $l$
19:      $p$.abs-dist$:= FW(p'$s function name$) + l$.abs-dist
20:    end if
21:    $VS := VS \cup \{p\}$
22:    $Q := Q \cup \{p\}$
23:  end for
24: end while
```

Figure 3.9: Algorithm: `abs-dist`

FW(p's function name)) and the l's abs-dist. The algorithm put p to VS and Q for further iterations. In contrast to the algorithm `exit-dist`, the algorithm `abs-dist` put a function entry location to Q because the algorithm annotates exit-dist across functions.

Handling Recursive Function Calls

If a program has recursive function calls, the `exit-dist` call for a function in the recursive function call chain is not available. For example, Figure 3.10 shows example function call dependency graphs. If a function x is a recursive function, the `exit-dist`(x) call is not available because x itself is the callee function of x. If more than one function makes up a recursive function call chain, the `exit-dist` call with any function in the chain is not available.

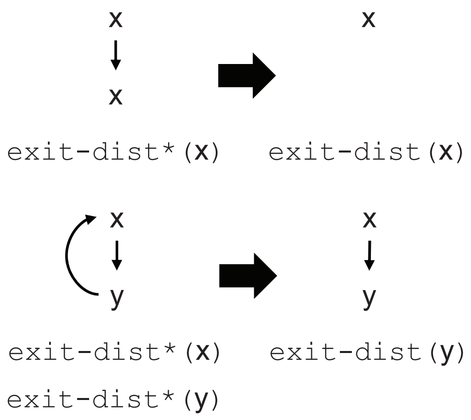


Figure 3.10: An example order of calling `exit-dist` and `exit-dist*` using function call dependency graph to handle recursive function calls

To annotate exit-dist for functions with recursive calls, I define the algorithm `exit-dist*`. The algorithm `exit-dist*` calculates the shortest distance of a function even though the function has a callee function that I do not know the shortest distance. Figure 3.11 shows the algorithm `exit-dist*`. The colored lines are differences between `exit-dist*` and `exit-dist`. The algorithm `exit-dist*` tries to calculate the shortest distance of a base case of the recursion by avoiding recursions. Thus, the algorithm `exit-dist*` skips a function return operation if FW does not have the shortest distance of the given function (line 18–19). The function of which the shortest distance is not defined by FW compose a recursive call chain. Since `exit-dist*` computes the shortest distance of a function not full annotation of exit-dist, the algorithm `exit-dist*` stops when it finds the function entry location (line 26–28, line 31–32).

As presented in Figure 3.10, the `exit-dist*`(x) call is available even if x has the callee function x. The `exit-dist*`(x) call computes the shortest distance of the base case of the given function. After that, the function x is removed from the dependency graph with allowing `exit-dist`(x) call. For a recursive call chain that consists of more than one function, you should call the `exit-dist*` for each function making up the call chain. And then the function with the shortest base case distance is removed from the dependency graph by breaking out the call chain as shown in Figure 3.10.

```
Input: $f$, the function name to compute exit-dist; Global Variables: $FW$, the mapping function from
    function name to the function's shortest distance;
 1: $l_{exit}$ := the function exit location of $f$
 2: $l_{entry}$ := the function entry location of $f$
 3: $l_{exit}$.exit-dist:= 0
 4: $VS := \emptyset$
 5: $Q := \emptyset$
 6: $VS := VS \cup \{l_{exit}\}$
 7: $Q := Q \cup \{l_{exit}\}$
 8: while $Q \neq \emptyset$ do
 9:   $l$ := pick a location with the smallest exit-dist from $Q$
10:   for all incoming operations $e$ of $l$ do
11:     $p$ := $e$'s source location
12:     if $p \in VS$ then
13:       continue
14:     end if
15:     if $e$ is not a function return operation then
16:       $p$.exit-dist:= $1 + l$.exit-dist
17:     else
18:       $p$ := the corresponding function call location of $l$
19:       if $FW(p'$s function name$) == nil$ then
20:         continue
21:       end if
22:       $p$.exit-dist:= $FW(p'$s function name$) + l$.exit-dist
23:     end if
24:     $VS := VS \cup \{p\}$
25:     $Q := Q \cup \{p\}$
26:     if $p == l_{entry}$ then
27:       $stop := TRUE$
28:       break
29:     end if
30:   end for
31:   if $stop$ then
32:     break
33:   end if
34: end while
35: $FW(f) := l_{entry}$.exit-dist
```

Figure 3.11: Algorithm: `exit-dist*`

3.2.2 Distance calculation using function call context tracking

TOUR calculates abs-dist and rel-dist of an abstract state using the annotated information (i.e., abs-dist and exit-dist') of the program locations and a function call context at runtime. The abs-dist of an abstract state is the same as the abs-dist of the corresponding program location. The rel-dist value of an abstract state is the sum of the exit-dist' of the corresponding location and the distance of the function call of the abstract state. The distance of an abstract state is the smaller distance of abs-dist and rel-dist.

TOUR tracks the function call context (i.e., the corresponding function call abstract state) in each abstract state. TOUR assigns the function call abstract state of an abstract state s' ($\texttt{call}(s')$) as follows using the predecessor abstract state s and the program operation op between s and s' :

- If s' is the initial abstract state (no successor abstract state s and no op)

$$\texttt{call}(s') = nil$$

- If op is a function call operation

$$\texttt{call}(s') = s$$

- If op is a function return operation

$$\texttt{call}(s') = \texttt{call}(\texttt{call}(s))$$

- Otherwise

$$\texttt{call}(s') = \texttt{call}(s)$$

Finally, the distance of an abstract state $s = \langle l, a\rangle$ (i.e., $\texttt{dist}(s)$) is calculated as follows:

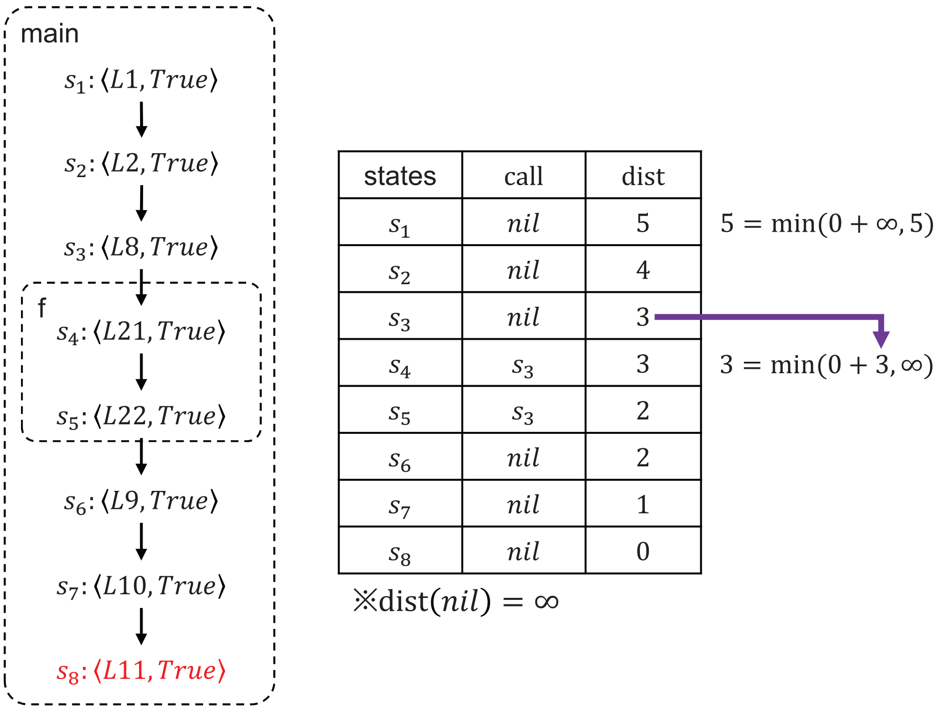
$$\texttt{dist}(s) = \begin{cases} \texttt{min}(l.\textsf{exit-dist'} + \texttt{dist}(\texttt{call}(s)), l.\textsf{abs-dist}) & s \neq nil \\ \infty & s = nil \end{cases} \tag{3.1}$$

Note that $l.\textsf{exit-dist'} + \texttt{dist}(\texttt{call}(s))$ denotes the s' rel-dist and $l.\textsf{abs-dist}$ denotes the s' abs-dist.

Figure 3.12 shows an example abstract state graph constructed by the directed ARMC for the example program in Figure 3.1. The directed ARMC uses exit-dist' presented in Figure 3.5 and abs-dist presented in Figure 3.6. The distance of an abstract state s is calculated as Equation 3.1.

The constructed abstract state graph shows the path with the shortest distance to the error location (L11) in terms of the number of edges as the distance. The function call abstract state of the initial abstract state s_1 is nil because no function invokes the function main that includes the program start location L1. The rel-dist of the abstract state $s_1 = \langle L1, True\rangle$ is the exit-dist' of L1 (i.e., 0) plus the distance of the function call abstract state of the function main (i.e., $\texttt{dist}(nil)= \infty$). The abs-dist of the abstract state $s_1 = \langle L1, True\rangle$ is the abs-dist of L1 (i.e., 5). The distance of s_1 is 5, which is the abs-dist of s_1 and the minimum of rel-dist (∞) and abs-dist (5).

The function main calls the function f with the abstract state s_4 at the abstract state s_3 corresponding to the location L8. The rel-dist of the abstract state $s_4 = \langle L21, True\rangle$ is the exit-dist' of L21 (i.e., 0) plus the distance of the function call abstract state of s_4 (i.e., s_3, which has the distance as 3). The abs-dist of the abstract state $s_4 = \langle L21, True\rangle$ is the same as the abs-dist of L21 (i.e., ∞). The distance of s_4 is 3, which is the rel-dist of s_4 and the minimum of rel-dist (3) and abs-dist (∞).


main
s_1: $\langle L1, True\rangle$
s_2: $\langle L2, True\rangle$
s_3: $\langle L8, True\rangle$
f
s_4: $\langle L21, True\rangle$
s_5: $\langle L22, True\rangle$
s_6: $\langle L9, True\rangle$
s_7: $\langle L10, True\rangle$
s_8: $\langle L11, True\rangle$
$5 = \min(0 + \infty, 5)$
$3 = \min(0 + 3, \infty)$
※dist(nil) = ∞

states	call	dist
s_1	nil	5
s_2	nil	4
s_3	nil	3
s_4	s_3	3
s_5	s_3	2
s_6	nil	2
s_7	nil	1
s_8	nil	0

Figure 3.12: An example abstract state graph with tracked function call context and calculated distance

The Directed ARMC Algorithm of TOUR

Figure 3.13 shows the shortest-distance first directed ARMC algorithm of TOUR. The highlighted lines are the differences between the directed ARMC algorithm and the ARMC algorithm in Figure 3.3. The algorithm takes an abstract state graph consists of a set of graph nodes N and a set of graph edges E; L_{err}, a set of error locations; `call`, a mapping function from an abstract state to the function call abstract state; and `dist`, a mapping function from an abstract state to the distance of the state.

The algorithm initializes the set of reached abstract states reached and the set of unmarked reached abstract states waitlist (line 1–2). The algorithm iterates until waitlist is empty (line 3). At the beginning of each iteration, the algorithm selects the abstract state with the shortest distance from waitlist (line 4). For the selected abstract state, the algorithm constructs a successor abstract state for each outgoing operations *op* of the selected abstract state (line 5–6). After constructing a successor abstract state $\langle l', a'\rangle$, the algorithm assigns the function call abstract state of $\langle l', a'\rangle$ (i.e., `call`$(\langle l', a'\rangle)$) according to the type of *op* (line 7–13). If *op* is a function call operation, the function call abstract state of $\langle l', a'\rangle$ is $\langle l, a\rangle$, i.e., the selected abstract state (line 7–8). If *op* is a function return operation, the function call abstract state of $\langle l', a'\rangle$ is `call`(`call`$(\langle l, a\rangle)$), i.e., the function call abstract state of $\langle l, a\rangle$'s function call abstract state (line 9–10). If *op* is not a function call or return operation, the function call abstract state of $\langle l', a'\rangle$ is `call`$(\langle l, a\rangle)$, i.e.,the same function call abstract state of the predecessor abstract state (line 12). After assigning the appropriate function call abstract state of $\langle l', a'\rangle$, the algorithm assigns the distance of $\langle l', a'\rangle$ using the Equation 3.1 (line 14). Note that the mapping function `dist` utilizing the recursion always terminates because the function call abstract state of the initial abstract state is *nil* (see Section 3.2.2).

The extended version of the directed ARMC algorithm of TOUR for a ARMC technique (i.e., block-abstraction memoization for interprocedural program analysis) is presented in Appendix (see Section 7.1).

Input: a set of abstract state graph nodes N; a set of abstract state graph edges E; a set of error locations L_{err}; **Global Variables:** `call`, a mapping function from an abstract state to the function call abstract state; `dist`, a mapping function from an abstract state to the distance of the state
Output: $safe$ if the graph contains no target path or $\langle unsafe, N, E\rangle$ if the graph contains a target path

```
1: reached := $N$
2: waitlist := unmarked nodes from $N$
3: while waitlist$\neq \emptyset$ do
4:   $\langle l, a\rangle$ := pop an abstract state with the shortest distance from waitlist
5:   for all outgoing operations op of $l$ do
6:     $\langle l', a'\rangle$ := $\langle l, a\rangle$'s successor abstract state according to op
7:     if op is a function call operation then
8:       call($\langle l', a'\rangle$) := $\langle l, a\rangle$
9:     else if op is a function return operation then
10:      call($\langle l', a'\rangle$) := call(call($\langle l, a\rangle$))
11:    else
12:      call($\langle l', a'\rangle$) := call($\langle l, a\rangle$)
13:    end if
14:    dist($\langle l', a'\rangle$) := min($l'$.exit-dist + dist(call($\langle l', a'\rangle$)), $l'$.abs-dist)
15:    reached := reached $\cup \{\langle l', a'\rangle\}$
16:    $E := E \cup \{\langle l, a\rangle \rightarrow \langle l', a'\rangle\}$
17:    if $a' \neq False$ then
18:      if $l' \in L_{err}$ then
19:        return $\langle unsafe$, reached, $E\rangle$
20:      end if
21:      waitlist := waitlist $\cup \{\langle l', a'\rangle\}$
22:    end if
23:  end for
24:  mark $\langle l, a\rangle$
25: end while
26: return $safe$
```

Figure 3.13: The directed ARMC algorithm

Chapter 4. Program-specific Distance Metric Selection

Figure 4.1 shows the graphical overview of the TOUR with the program-specific distance metric selection. I consider the distance metric selection and the model generation (i.e., the part in the dashed region) in this chapter. The program annotation and the directed model checking are dealt in Section 3.2.

I propose four different distance metrics for TOUR because the best distance metric is probably different for each ARMC technique. I propose the distance metrics considering which program statements mainly affect the performance of each ARMC technique. I also extend the program annotation algorithms (the algorithms `exit-dist` and `abs-dist` in Section 3.2.1) to generalize the algorithm for various distance metrics.

I also think that the best distance metric is different for each program, even if a distance metric is generally good for an ARMC technique. Thus, I propose a method to generate a model for program-specific distance metric selection to select the best distance metric among the four distance metrics for a target program. TOUR generates such a model by learning from historical directed ARMC results with static program features.

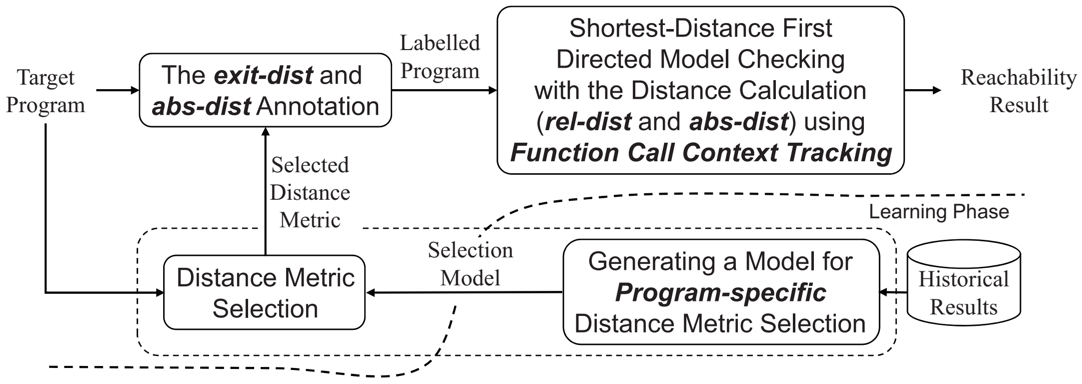

Target Program
The ***exit-dist*** and ***abs-dist*** Annotation
Labelled Program
Shortest-Distance First Directed Model Checking with the Distance Calculation (***rel-dist*** and ***abs-dist***) using ***Function Call Context Tracking***
Reachability Result
Selected Distance Metric
Learning Phase
Selection Model
Distance Metric Selection
Generating a Model for ***Program-specific*** Distance Metric Selection
Historical Results

Figure 4.1: Overview of TOUR with the program-specific distance metric selection

4.1 Distance Metrics

The four distance metrics proposed for TOUR are a number of statements (st), a number of basic blocks (bb), a number of loop heads (lh), and a number of loop heads and function entry/exit points (lf):

- **The st metric** considers each statement equally and is widely used in model checking or symbolic execution [23, 27, 28].
- **The bb metric** weighs branch condition statements to focus on the number of basic blocks instead of the number of statements. It is effective for programs including many statements that hardly affect ARMC technique performance, e.g., an assignment statement with no operation.
- **The lh metric** weighs loop branch condition statements. It is effective for ARMC techniques that compute an abstract state for a large number of acyclic statements at once, e.g., block encoding [13, 12].

- **The lf metric** weighs loop branch condition statements and the function call and return statements. It is effective for ARMC techniques that explore each function's search space separately, e.g., block-abstraction memoization [40, 14].

To generalize TOUR for various distance metrics, I extend the program annotation algorithms presented in Section 3.2.1. Figure 7.7 in Section 7.3 of Appendix shows the extended exit-dist annotation algorithm, and Figure 7.8 in Section 7.3 of Appendix shows the extended abs-dist annotation algorithm. The shaded line shows the difference between the original algorithm and the extended algorithm. Both algorithms use the additional mapping function W that indicates the edge weight of a given control-flow edge. The extended algorithms use W to annotate the value to the predecessor location if the operation (control-flow edge) e is not a function return operation while the original algorithms annotate the value of the predecessor location as the value of the current location l plus one (line 16 of both algorithms). The mapping function W is the generalized version of edge weights. You can determine how much edge weight is assigned to which type of edge in W according to a distance metric.

The edge weight function W of an edge e for each proposed distance metric is defined as follows:

- For a number of statements (st)

$$W(e) = 1 \tag{4.1}$$

- For a number of basic blocks (bb)

$$W(e) = \begin{cases} 1, & \text{if } e \text{ is a condition operation} \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

- For a number of loop heads (lh)

$$W(e) = \begin{cases} 1, & \text{if } e \text{ is a condition operation corresponding} \\ & \text{to a loop condition} \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

- For a number of loop heads and function entry/exit points (lf)

$$W(e) = \begin{cases} 1, & \text{if } e \text{ is a condition operation corresponding} \\ & \text{to a loop condition} \\ 1, & \text{if the predecessor of } e \text{ is a function entry} \\ & \text{location} \\ 1, & \text{if } e \text{ is a function return operation} \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

4.2 Generating a Model for Program-specific Distance Metric Selection

TOUR generates a model for program-specific distance metric selection by learning from historical results consisting of determining the best distance metric and static feature value of each program. A generated model selects the best distance metric among the four distance metrics by regarding the static program feature values.

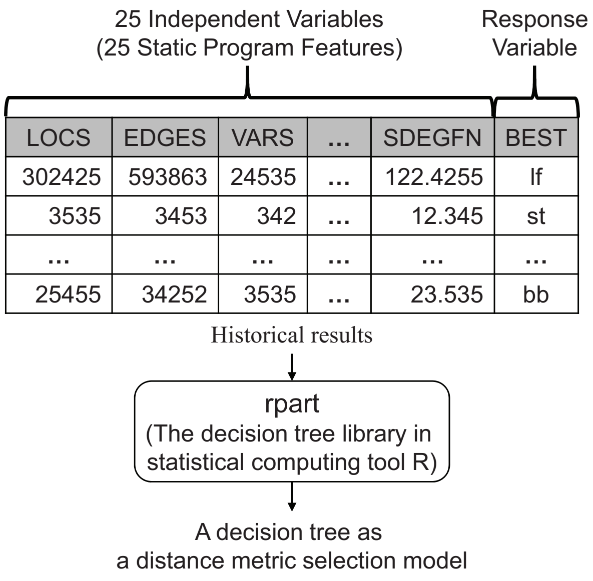


Figure 4.2: The procedure for generating a model for program-specific distance metric selection from historical results

Figure 4.2 shows the procedure for generating a program-specific distance metric selection model from historical results. The historical directed model checking results are composed of 25 static feature values and the best distance metric of each program. The 25 features represent size (Locations and Edges categories), modularity (Loops and Functions categories), data flow (Variables category), and complexity (Cyclomatic Complexity [60] category), as shown in Table 4.1. For features defined for a single function, TOUR uses the maximum, average, and standard deviation values of functions in a program as the representative values. TOUR label each program in the historical results with its best distance metric in terms of the CPU time among the four classes, st, bb, lh, and lf. The detailed demonstration of how to calculate the 25 static program features is presented at the Appendix chapter (see Section 7.2).

TOUR uses the recursive partitioning algorithm (the rpart library in open source statistical computing tool R [61]) to build a decision tree model. TOUR uses the recursive partitioning algorithm because it is widely used for multiclass classification over the past three decades and is known as accurate.

As a result of optimization, TOUR filters out historical results of less than 40 seconds of CPU time difference caused by the distance metric change. Figure 4.3 shows the example of how TOUR filters out historical results. Each row denotes each program in the historical results. The CPU time columns denote the model checking time of historical directed model checking results using each distance metric. The BEST column indicates the best distance metric for the program in terms of the CPU time of each directed ARMC. For example, BEST of the program corresponding to the first row is st because the CPU time is the smallest when using st as the distance metric for the ARMC of the program. I compute the CPU time difference between the best distance metric and the worst distance metric (i.e., the max-min column) to filter out programs from model generation. For example, max-min of the first column is 80 (=100-20) by comparing lh and st. Suppose the filter-out threshold is 10 seconds, the second row of Figure 4.3 is filtered out from the model generation. If there is a failed ARMC of a program such as time-out, memory-out, as shown in the last row of Figure 4.3, TOUR assigns max-min as ∞ to be included in the model generation.

Table 4.1: Category and description of the 25 program features used for distance metric selection rule generation; Cyclomatic Complexity: the well-known McCabe's cyclomatic complexity [62]

Category	Feature name	Description
Locations	LOCS	Number of program locations
	MXLOFN	Maximum number of locations per function
	AVLOFN	Average number of locations per function
	SDLOFN	Standard deviation of the number of locations per function
Edges	EDGES	Number of control-flow edges
	MXEGFN	Maximum number of control-flow edges per function
	AVEGFN	Average number of control-flow edges per function
	SDEGFN	Standard deviation of the number of control-flow edges per function
Loops	LOOPS	Number of loops in a program
	MXLPFN	Maximum number of loops per function
	AVLPFN	Average number of loops per function
	SDLPFN	Standard deviation of the number of loops per function
Functions	FUNCS	Number of functions in a program
	MXCALLS	Maximum number of function calls per function
	AVCALLS	Average number of function calls per function
	SDCALLS	Standard deviation of the number of function calls per function
Variables	VARS	Number of relevant variables (used in conditions and their dependent variables)
	VARSASM	Number of variables used in conditions
	VARSLOOP	Number of variables used in loop exit conditions
	VARSINC	Number of variables used as increasing/decreasing count variables for `for` statements
	FIELDS	Number of relevant bit-field variables
Cyclomatic Complexity	MXCC	Maximum cyclomatic complexity per function
	SMCC	Sum of cyclomatic complexity per function
	AVCC	Average cyclomatic complexity per function
	SDCC	Standard deviation of cyclomatic complexity per function

CPU time (s) (used distance metric)				BEST	max-min
st	bb	lh	lf		
20	32	100	24	st	80
20	19	15	17	lh	5
...	...	...	...	...	...
TO	18	20	21	bb	∞

Figure 4.3: Preprocessing the historical results of directed model checking; TO, time-out; the shaded row, filtered out by min-max threshold 10 seconds

4.3 Distance Metric Selection

TOUR generates a decision tree as a selection model by learning from the historical results. Figure 4.4 shows an example decision tree to select a distance metric. Each internal node represents a static feature name and the decision condition for the feature to select a child node (i.e., select the left child node if the decision condition is satisfied). Each terminal node represents the selected distance metric. For example, suppose a program has static feature values as FUNCS $= 840$, VARSINC $= 11$, AVLOFN $= 39$, and MXCC $= 30$. According to the decision tree in Figure 4.4, TOUR selects the left, right, right, right, and left child from the root to the leaf. Finally, TOUR selects bb as the distance metric for the program.

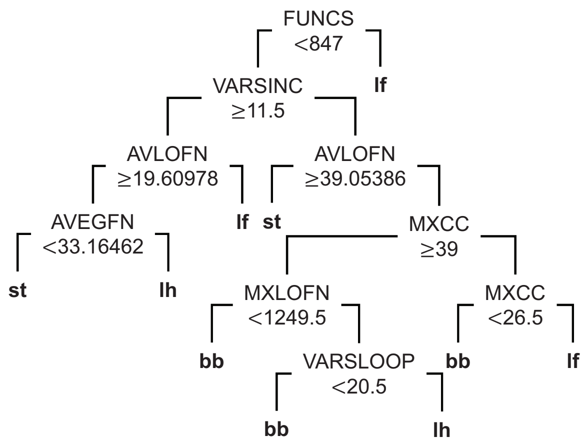


Figure 4.4: An example distance metric selection model represented as a decision tree

Chapter 5. Empirical Evaluation

5.1 Experimental Setup

5.1.1 Research Questions

- **RQ1. Effectiveness of TOUR on bug detection ability:** Does TOUR improve the bug detection ability of ARMC?

  I investigate whether TOUR finds bugs in more real-world buggy C programs than existing ARMC techniques.

- **RQ2. Effectiveness of TOUR on verification speed:** Does TOUR save the verification time of ARMC?

  I investigate whether TOUR saves CPU time with two different existing ARMC techniques for real-world clean C programs.

- **RQ3. Effectiveness of the proposed distance metrics:** Which metric among the four distance metrics is the best distance metric for each ARMC technique?

  I investigate which distance metric is the best in terms of the number of solved programs and CPU time among the four distance metrics for each ARMC technique.

- **RQ4. Effectiveness of the distance metric selection:** Does TOUR select the best distance metric for each target program?

  I investigate whether TOUR using the program-specific metric selection model takes less CPU time than TOUR using a single metric for target programs.

- **RQ5. Effectiveness of the historical result filtering:** Dose the historical result filtering is effective for generating a better selection model?

  I investigate whether TOUR using the historical result filtering shows better performance than TOUR using no filtering. I also investigate which filtering threshold is the best for each ARMC technique.

5.1.2 Techniques to Compare

I implemented TOUR in a form integrated into two state-of-the-art ARMC techniques, i.e., lazy abstraction and block-abstraction memoization. By applying TOUR on *the most widely used optimization method*, i.e., lazy abstraction [16, 17, 12, 14, 15, 33, 42, 28], I investigate whether TOUR is generally effective in speeding up ARMC. I also investigate whether TOUR improves the practical performance of ARMC for practitioners. Thus, I investigate how much TOUR improves the bug detection ability and saves the model checking time of *a current state-of-the-art ARMC technique*, i.e., block-abstraction memoization [14].

- **LA**: Lazy Abstraction (LA) is an ARMC technique that applies different abstractions for different substate graphs [9]. Thus, LA can eliminate a false alarm by refining only the substate graph relevant to the false alarm. LA uses the depth-first search. I use LA implemented in the open-source model checking tool CPAchecker [63] (CPAchecker 2.1). LA uses MathSAT5 [65] (MathSAT5 version 5.6.5 (63ef7602814c)) as the SMT-solver for constraint solving. I used LA for RQ1 and RQ2 as a baseline ARMC technique.

- **LA.T**: LA.T is the technique that applies TOUR to LA. LA.T uses the default parameter values of the rpart library except two parameters `maxcompete` = 0 and `maxsurrogate` = 0 to generate a selection model [61]. LA.T generates a distance metric selection model by excluding the target program from the historical result to avoid over-fitting. I used LA.T for RQ4.

- **BAM**: Block-Abstraction Memoization for interprocedural program analysis (BAM) is a current state-of-the-art ARMC technique [14]. It extends block-abstraction memoization [40] to handle recursive programs. It combines lazy abstraction [9], predicate abstraction, explicit-value abstraction [15], large-block encoding [12], and block-abstraction memoization. I use BAM implemented in the open-source model checking tool CPAchecker [63] (CPAchecker 2.1). While adopting the depth-first search, BAM uses SMTInterpol [66] (SMTInterpol 2.5-732-gd208e931) as the SMT solver for constraint solving. I used BAM for RQ1 and RQ2 as a baseline ARMC technique.

- **BAM.T**: BAM.T is the technique that applies TOUR to BAM. BAM.T uses the same parameter setting of LA.T for rpart to generate a selection model. BAM.T generates a distance metric selection heuristic by excluding the target program from the historical result to avoid overfitting. I used BAM.T for RQ4.

- **LA.T.{st,bb,lh,lf}**: LA.T.{st,bb,lh,lf} are the techniques that apply TOUR using a single distance metric to LA without distance metric selection. I compare LA.T.{st,bb,lh,lf} in RQ1, RQ2, RQ3 and RQ4.

- **BAM.T.{st,bb,lh,lf}**: BAM.T.{st,bb,lh,lf} are the techniques that apply TOUR using a single distance metric to BAM without distance metric selection. I compare BAM.T.{st,bb,lh,lf} in RQ1, RQ2, RQ3 and RQ4.

- **LA.T.{NO,10,20,30,40,50}**: LA.T.{NO,10,20,30,40,50} are techniques that generates a distance metric selection model using {NO,10,20,30,40,50} seconds of the historical result filtering threshold. I used LA.T.{NO,10,20,30,40,50} in RQ5.

- **BAM.T.{NO,10,20,30,40,50}**: BAM.T.{NO, 10, 20, 30, 40, 50} are techniques that generates a distance metric selection model using {NO, 10, 20, 30, 40, 50} seconds of the program filtering threshold. I used BAM.T.{NO, 10, 20, 30, 40, 50} in RQ5.

5.1.3 Target programs

- **RealWorld**: RealWorld is the set of C programs obtained from the SoftwareSystems category of SV-COMP '21 [67, 64]. The category contains 3,184 programs from *real-world sources*, and I exclude programs with no predefined error location. The programs consists of Linux device driver programs [68], Amazon AWS C commons library software, and uthash hashing library software, which are real-world software rather academic. As a result, RealWorld includes 3,042 programs

Table 5.1: The statistics of RealWorld.{1,2,3,4}; SMCC: the sum of Cyclomatic Complexity per function

Index	SMCC	Number of Programs
1	$[24, 87)$	748
2	$[87, 448.5)$	773
3	$[448.5, 1412.5)$	760
4	$[1412.5, 132302]$	761

consisting of 2,688 clean programs and 354 buggy programs. I used RealWorld for all experiments. Additionally, I divided RealWorld into four subsets according to the complexity of programs.

- **RealWorld.{1,2,3,4}**: RealWorld.{1,2,3,4} are the subsets of RealWorld divided by the program complexity. I use *the SuM of the Cyclomatic Complexity per function* (SMCC) of a program to represent the complexity of the program. Cyclomatic complexity is a well-known complexity metric for programs written in an imperative programming language such as C [62]. I divided RealWorld into four subsets considering quantiles. Table 5.1 presents the boundary values and the number of programs in each subset. I used RealWorld.{1,2,3,4} in the experiment for RQ2 to show the increasing effectiveness of TOUR on verification speed as the complexity of programs increases.

These RealWorld programs are used for experiments because many recent model checking studies use them as verification targets [32, 33, 69, 59, 46]. In particular, BAM participated in SV-COMP 21' and *showed the best verification performance* (i.e., solving the largest number of verification tasks) among other participating techniques for the SoftwareSystems category. Note that the limitation of using target programs with predefined error locations is acceptable because various studies in the verification and testing fields evaluate their technique on programs with assertion statements, which are a type of predefined error location [70, 71, 38, 39, 26].

5.1.4 Measurement

I measure *CPU time* and *the number of solved programs* for each technique to compare. Since I limit the model checking time for each target program, not only the CPU time but also the number of solved programs shows the time efficiency of a technique.

- **CPU time**: I measure the CPU time of a model checking run, which includes the CPU time of all subprocesses that the main process invokes. The CPU time of TOUR includes the time for extracting program features. I exclude the time for generating a distance metric selection heuristic from the CPU time because a generated heuristic is used for multiple target programs. The CPU time is measured by the benchmark execution tool BENCHEXEC (see Section 5.1.5).

- **The number of solved programs** I count the number of correctly solved programs of a technique. A technique correctly solves a buggy program if it finds a bug for the program within a resource limit. A technique correctly solves a clean program if it finishes model checking without a bug or a false alarm within a resource limit. A technique fails to solve a target program if it incorrectly solves or fails to finish due to the time-out, memory-out, or internal error of the technique.

5.1.5 Model Checking Environmental Setup

I conducted all the model checking runs by using BENCHEXEC 3.6 [72] (i.e., a benchmark execution platform used in SV-COMP '21) to ensure reliable experimental results. I used five machines with a 3.4GHz CPU and 16 GB memory for all model checking runs. All machines used Ubuntu 20.04 and OpenJDK 1.13. The resource limit for a model checking run was four CPU cores, 15 GB RAM, and up to 900 seconds of CPU time. The 15 GB RAM and up to 900 seconds of CPU time are the same resource limits of SV-COMP '21 which is considered standard [67].

5.1.6 Threats to validity

- **Internal validity:** A threat to internal validity is possible bugs in TOUR implementation and the other techniques I studied. I meticulously verified our implementations to address this threat.

  Additionally, I controlled the model checking execution environment using machines with the same specification and operating environment. Furthermore, I conducted the experiments using the benchmarking tool BENCHEXEC used in SV-COMP '21. BENCHEXEC isolates the model checking executions from the interruption of other processes executed in the same machine to obtain reliable results. Thus, I believe that the threat to internal validity is limited.

- **External validity:** A threat to external validity is the representativeness of our target programs. I expect that this threat is limited since the target programs are widely used benchmark programs and tested by many other researchers.

  I used target programs that represent general real-world C programs. RealWorld includes programs obtained from real-world software projects (e.g., Linux-device driver programs), and they cover most of the C features. Thus, I believe that the threat to external validity is also limited.

- **Construct validity:** I used two measures, i.e., the number of solved programs and CPU time. These measures have been widely and generally used performance criteria for model checking in recent studies [44, 45, 16, 59, 73, 28]. Thus, I believe that the threat to construct validity is limited.

5.1.7 Limitations

I do not evaluate TOUR on multi-thread programs. Despite TOUR is proposed for fast reachability analysis, it currently does not support the reachability on multi-threaded programs [74].

I do not evaluate TOUR on verifying a termination property [75, 76]. A path that does not terminate is a target path in a termination property verification. It is difficult to define a target location for the not-terminated path and there is no known definition to define the target location. Thus, the current implementation of TOUR only supports an explicit unsafe program state as a search target.

I do not evaluate TOUR on memory safety verification [77, 78]. TOUR is applicable to memory safety verification because a unsafe state of the memory safety property can be presented as an explicit error location. However, the conversion of the memory safety property into an explicit error location needs an additional program instrumentation, which is out-of-scope of this study.

Table 5.2: The number of solved buggy RealWorld programs and the total CPU time of BAM and BAM.T; Count.Solved: the number of correctly solved programs; Count.Time-out: the number of programs that a technique fails due to time-out; Count.Others: the number of programs that a technique fails due to reasons other than time-out; Total CPU time: the total model checking time for 354 buggy RealWorld programs

	BAM	BAM.T	
Count.Solved	100	120	(↑ **20**%)
Count.Time-out	152	131	(↓ **14**%)
Count.Others	102	103	
Total CPU-time	154, 555s	136, 856s	(↓ **11**%)

5.2 Experimental Result

5.2.1 RQ1: Effectiveness of TOUR on bug detection ability

The results show that TOUR *meaningfully improves the bug detection ability* of an ARMC technique. BAM.T finds bugs in 20% (= (120-100)/100) more programs than BAM for the 354 buggy programs within 11% (= (154,555-136,856)/154,555) less total CPU time, as shown in Table 5.2. LA.T finds bugs in 118% (= (98-45)/45) more programs than LA for the 354 buggy programs within 3% (= (126,043-121,921)/121,921) more total CPU time as shown in Table 5.3.

The reason for the increased number of solved buggy programs caused by TOUR in BAM is the reduced number of failed results caused by time-out. Table 5.2 shows the number of buggy RealWorld programs that BAM.T and BAM fail due to time-out. BAM.T timed out for 131 (↓ 14%) programs, while BAM timed out for 152.

The seemingly slower speed (the 3% (= (126,043-121,921)/121,921) more total CPU time) of LA.T is due to the weakness of LA that does *not* support recursive function calls. In contrast to LA, which terminates as soon as possible when it observes a recursive execution, LA.T spends more time avoiding the recursive execution to detect a target path that does not include recursive executions. As a result, LA.T reduces the number of failed results caused by the recursion (i.e., reduced 65% (= (78-27)/78) compared to LA), as shown in Table 5.3.

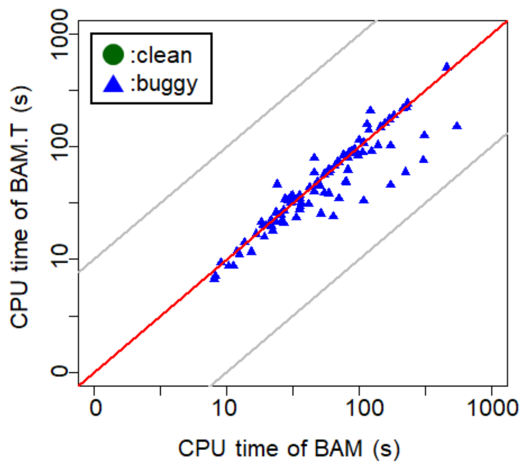
Table 5.3: The number of solved buggy RealWorld programs and the total CPU time of LA and LA.T; Count.Solved: the number of correctly solved programs; Count.Time-out: the number of programs that a technique fails due to time-out; Count.Error(recursion): the number of programs that a technique fails due to the recursive function calls; Count.Others: the number of programs that a technique fails due to reasons other than time-out and recursion; Total CPU time: the total model checking time for 354 buggy RealWorld programs

	LA	LA.T	
Count.Solved	45	98	(↑ **118**%)
Count.Time-out	99	94	
Count.Error(recursion)	78	27	(↓ **65**%)
Count.Others	132	135	
Total CPU-time	121, 921s	126, 043s	(↑ 3%)

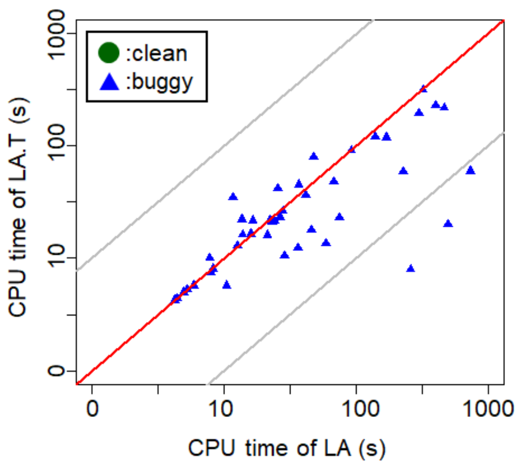
Comparison on Target Programs that Both Baseline and TOUR Techniques Find Bugs

Additionally, I compare a baseline technique to a TOUR technique program by program to evaluate the improved efficiency of TOUR regardless of ARMC failures. I utilize the scatter plot analysis to show CPU time of both techniques program by program.

BAM.T shows 13 seconds less CPU time than BAM on average for 97 RealWorld buggy programs that both BAM.T and BAM find bugs. Figure 5.1a shows the scatter plot of CPU time for 97 RealWorld buggy programs that both BAM.T and BAM find bugs. The x-axis denotes CPU time of BAM, and the y-axis denotes CPU time of BAM.T. Both axes are in log-scale. The red-line means that BAM.T and BAM take the same CPU time. The grey-line means that one technique is ten times faster than the other technique. The points are placed close to the x-axis showing less CPU time of BAM.T compared to BAM.



(a)



(b)

Figure 5.1: The scatter plot of a baseline technique's CPU time (x-axis) and a TOUR technique's CPU time (y-axis) in log-scale (a) for 97 RealWorld buggy programs that both BAM.T and BAM find bugs; (b) for 42 RealWorld buggy programs that both LA.T and LA find bugs

LA.T shows 54 seconds less CPU time than LA on average for 42 RealWorld buggy programs that both LA.T and LA find bugs. Figure 5.1b shows the scatter plot of CPU time for 42 RealWorld buggy programs that both LA.T and LA find bugs. The x-axis denotes CPU time of LA, and the y-axis denotes CPU time of LA.T. Both axes are in log-scale. The red-line means that LA.T and LA take the same CPU time. The grey-line means that one technique is ten times faster than the other technique. The points are placed close to the x-axis showing less CPU time of LA.T compared to LA.

5.2.2 RQ2: Effectiveness of TOUR on verification speed

The results show that TOUR *meaningfully improves the verification speed* of an ARMC technique. BAM.T verifies 15% (= (378-330)/330) more programs in 15% (= (336,940-287,911)/336,940) less total CPU-time than BAM for the 652 clean programs in RealWorld.4, i.e., the most complex program group, as shown in Table 5.4. I focus on complex target programs (i.e., RealWorld.4) because it is more meaningful to solve challenging programs quickly than to solve easy programs quickly.

The reason for the increased number of solved clean programs caused by TOUR in BAM is the

Table 5.4: The number of solved clean RealWorld.4 programs and the total CPU time of BAM and BAM.T; Count.Solved: the number of correctly solved programs; Count.Time-out: the number of programs that a technique fails due to time-out; Count.Others: the number of programs that a technique fails due to reasons other than time-out; Total CPU-time: the total model checking time for the 652 clean RealWorld.4 programs

	BAM	BAM.T	
Count.Solved	330	378	(↑ **15**%)
Count.Time-out	316	268	(↓ **15**%)
Count.Others	6	6	
Total CPU time	336, 940s	287, 911s	(↓ **15**%)

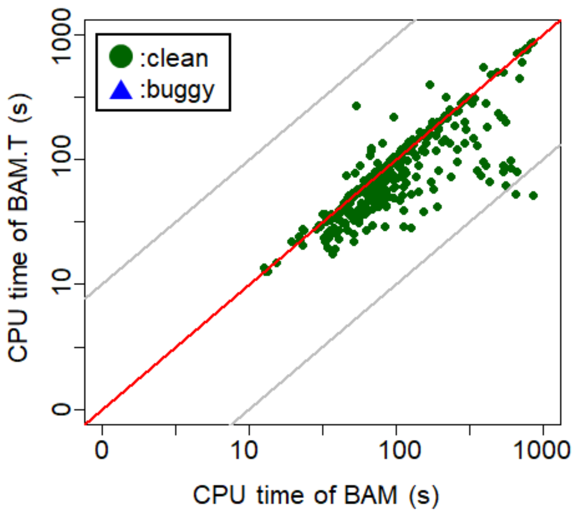


Figure 5.2: The CPU time of BAM (x-axis) and BAM.T (y-axis) in log-scale for 323 clean RealWorld.4 programs that both BAM and BAM.T verify

reduced number of failed results caused by time-out. Table 5.4 shows the number of clean programs that BAM and BAM.T fail due to time-out. BAM.T reduces the number of failed results caused by timeout by 15% (= (316-268)/316) compared to BAM for the complex programs.

Additionally, I compare BAM.T and BAM program by program to evaluate the improved efficiency of TOUR regardless of ARMC failures. I utilize the scatter plot analysis to show CPU time of both techniques program by program.

BAM.T takes less CPU time than BAM 216 out of 323 clean RealWorld.4 programs that both BAM.T and BAM verify. Also, BAM.T takes 36 seconds less CPU time than BAM on average for the 323 clean RealWorld.4 programs that both BAM.T and BAM verify. Figure 5.2 shows the scatter plot of CPU time for 97 RealWorld buggy programs that both BAM.T and BAM find bugs. The x-axis denotes CPU time of BAM, and the y-axis denotes CPU time of BAM.T. Both axes are in log-scale. The red-line means that BAM.T and BAM take the same CPU time. The grey-line means that one technique is ten times faster than the other technique. The points are placed close to the x-axis showing less CPU time of BAM.T compared to BAM.

Improved Verification Speed on the entire set of the 2688 Clean RealWorld Programs

The results show that TOUR improves the verification speed of an ARMC technique on the 2,688 clean RealWorld programs. BAM.T verifies 4% (= (1,996-1,911)/1,911) more programs in 12% (= (657,265-581,045)/657,265) less total CPU time than BAM for the 2,688 clean RealWorld programs as shown in Table 5.5.

Table 5.5: The number of solved programs and total CPU time of each technique for 2,688 clean RealWorld programs; C.SOLV: the number of programs solved by each technique; T.TIME: the total CPU-time of each technique

Technique	C.SOLV	T.TIME
BAM	$1,911$	$657,265$s
BAM.T	$1,996$ (↑ **4**%)	$581,045$s (↓ **12**%)
LA	$1,404$	$650,580$s
LA.T	$1,420$ (↑ **1**%)	$673,734$s (↑ 4%)

LA.T solves more programs than LA, but it takes more total CPU time than LA. LA.T verifies 1% (= (1,420-1,404)/1,404) more programs in 4% (= (673,734-650,580)/650,580) more total CPU time than LA for the 2,688 clean RealWorld programs as shown in Table 5.5. [1]

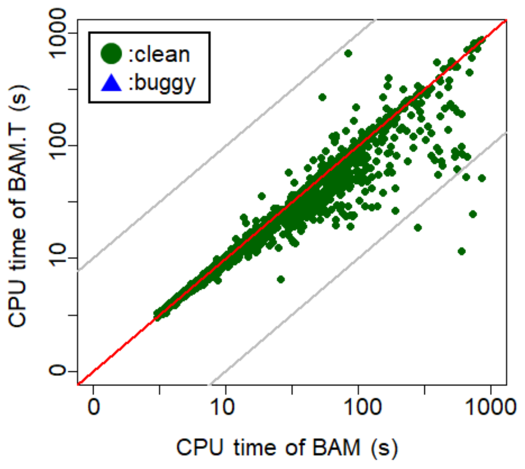
I utilize the scatter plot analysis to show CPU time of both techniques program by program. BAM.T takes less CPU time than BAM on 1084 out of 1900 clean RealWorld programs that both BAM.T and BAM verify. BAM.T takes 9 seconds less CPU time than BAM on average for 1900 clean RealWorld programs that both BAM.T and BAM verify. Figure 5.3a shows the scatter plot of CPU time for 1900 clean RealWorld programs that both BAM.T and BAM verify. The x-axis denotes CPU time of BAM, and the y-axis denotes CPU time of BAM.T. Both axes are in log-scale. The red-line means that BAM.T and BAM take the same CPU time. The grey-line means that one technique is ten times faster than the other technique. The points are placed close to the x-axis showing less CPU time of BAM.T compared to BAM.

LA.T takes less CPU time than LA on 559 out of 1398 clean RealWorld program that both LA.T and LA verify. Although LA.T takes much CPU time than LA for more programs, LA.T shows 3 seconds less CPU time than LA on average for the 1398 clean RealWorld program that both LA.T and LA verify. Figure 5.3b shows the scatter plot of CPU time for the 1398 clean RealWorld program that both LA.T and LA verify. The x-axis denotes CPU time of LA, and the y-axis denotes CPU time of LA.T. Both axes are in log-scale. The red-line means that LA.T and LA take the same CPU time. The grey-line means that one technique is ten times faster than the other technique. The points are placed close to the x-axis showing less CPU time of LA.T compared to LA.

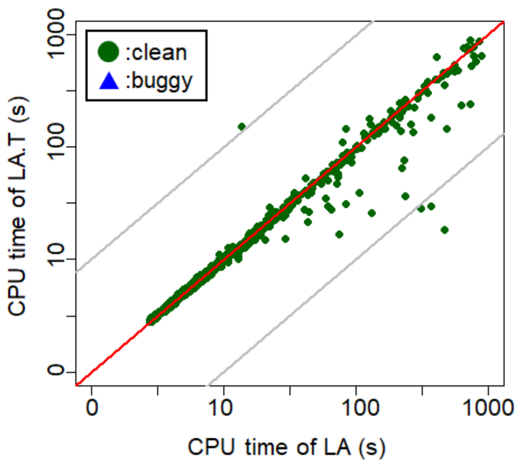
Increasing Effectiveness as Complexity Increases

The results show that the effectiveness of TOUR increases with the increasing complexity of clean programs. Table 5.6 shows the increasing effectiveness of TOUR (in parentheses) for clean RealWorld.{1,2,3,4} programs with the number of solved clean programs and the total model checking time of BAM.T. BAM.T verifies the same number of programs by taking less than 1% more total CPU-time than

[1] LA.T cannot verify a clean program by bypassing recursions that LA.T does for buggy programs.



(a)



(b)

Figure 5.3: The scatter plot of a baseline technique's CPU time (x-axis) and a TOUR technique's CPU time (y-axis) in log-scale (a) for 1900 clean RealWorld programs that both BAM.T and BAM verify; (b) for 1398 clean RealWorld programs that both LA.T and LA verify

Table 5.6: The number of solved programs and total CPU time of BAM.T for clean RealWorld.{1,2,3,4} programs; C.SOLV: the number of programs solved by BAM.T for each PROG.SET (difference ratio compared to that of BAM); T.TIME: the total CPU-time of BAM.T for each PROG.SET (difference ratio compared to that of BAM)

PROG.SET	C.SOLV		T.TIME	
RealWorld.1	691	(0%)	32,222s	(0%)
RealWorld.2	542	(↑ **1%**)	89,683s	(↓ **6%**)
RealWorld.3	385	(↑ **8%**)	171,229s	(↓ **11%**)
RealWorld.4	378	(↑ **15%**)	287,911s	(↓ **15%**)

BAM for RealWorld.1, i.e., the set of the least complex programs. BAM.T solves 15% more programs in 15% less total CPU-time than BAM for RealWorld.4, i.e., the set of the most complex programs.

Increasing effectiveness is not shown in LA for clean RealWorld.{1,2,3,4} programs because both LA and LA.T hardly verify complex programs, as shown in Table 5.7. LA.T verifies 2% more programs in 11% less total CPU-time than LA for clean RealWorld.2 programs. LA.T verifies 4% more programs within 3% more total CPU-time than LA for clean RealWorld.3 programs. For RealWorld.1, LA.T does not outperform LA because both LA.T and LA solve most of the target programs. For RealWorld.4, LA.T does not outperform LA because both LA.T and LA hardly solve the complex programs.

5.2.3 RQ3: Effectiveness of the proposed distance metrics

The results show that **lf** *is the best distance metric* in terms of both the number of solved programs and the total CPU time for BAM. For LA, **bb** *is the best distance metric* in terms of the number of solved programs. BAM.T.lf (i.e., the best distance metric in terms of both the number of solved programs and the total CPU time for BAM) solves 9% (= (2,096-1,923)/1,923) more programs in 21% (= (924,237-732,531)/924,237) less total CPU-time than BAM.T.st (i.e., the worst distance metric for BAM) as

Table 5.7: The number of solved programs and total CPU-time of LA.T for clean RealWorld.{1,2,3,4} programs; C.SOLV: the number of programs solved by LA.T for each PROG.SET (difference ratio compared to that of LA); T.TIME: the total CPU-time of LA.T for each PROG.SET (difference ratio compared to that of LA)

PROG.SET	C.SOLV		T.TIME	
RealWorld.1	685	(0%)	46,794s	(↑ 1%)
RealWorld.2	499	(↑ **2%**)	97,608s	(↓ **11%**)
RealWorld.3	175	(↑ **4%**)	230,153s	(↑ 3%)
RealWorld.4	61	(0%)	299,179s	(↑ 13%)

shown in Table 5.8. LA.T.bb (i.e., the best distance metric in terms of the number of solved programs for LA) solves 2% (= (1,500-1,471)/1,471) more programs in 12% (= (817,056-730,531)/730,531) more total CPU-time compared to LA.T.lh (i.e., the worst distance metric for LA) as shown in Table 5.9.

I compare each BAM.T with single distance metric to BAM. BAM.T.st solves 4% less programs within 14% more total CPU time than BAM, which shows worse performance even though BAM.T.st uses the error location directed ARMC. BAM.T.bb solves 2% more programs within 2% less total CPU time than BAM. BAM.T.lh solves 1% more programs within 3% less total CPU time than BAM. BAM.T.lf solves 4% more programs within 10% less total CPU time than BAM.

I also compare each LA.T with single distance metric to LA. LA.T.st solves 4% more programs within 7% more total CPU time than LA. LA.T.bb solves 4% more programs within 6% more total CPU time than LA. LA.T.lh solves 2% more programs within 5% less total CPU time than LA. LA.T.lf solves 3% more programs within 7% more total CPU time than LA.

In conclusion, the performance of a directed ARMC technique depends on the used distance metric. Also, a suitable distance metric varies with regard to the characteristics of an ARMC technique. It may necessary to develop a specific distance metric for an ARMC technique. Even if the directed ARMC techniques show generally better performance than the non-directed ARMC technique, the directed ARMC with a bad distance metric can be worse than the non-directed ARMC technique. A distance metric selection might effective to improve the performance of a directed ARMC. I assume the optimal selection case (BAM.T.opt and LA.T.opt) as a comparison criteria. BAM.T.opt solves 7% more programs within 19% less total CPU time than BAM, which shows significant improvement. Also, LA.T.opt solves 5% more programs within 15% less total CPU time than LA, which shows significant time reduction while BAM.T.{st,bb,lf} takes much time even if they solve more programs.

Table 5.8: The number of solved programs and total CPU time of BAM.T.{st,bb,lh,lf}; C.SOLV: the number of programs solved; T.TIME: The total CPU time for 3,042 (both buggy and clean) RealWorld programs

	C.SOLV		T.TIME	
BAM	2,011		811,819s	
BAM.T.st	1,923	(↓ 4%)	924,237s	(↑ 14%)
BAM.T.bb	2,044	(↑ **2%**)	791,910s	(↓ **2%**)
BAM.T.lh	2,030	(↑ **1%**)	789,454s	(↓ **3%**)
BAM.T.lf	2,096	(↑ **4%**)	732,531s	(↓ **10%**)

Table 5.9: The number of solved programs and total CPU time of LA.T.{st,bb,lh,lf}; C.SOLV: the number of programs solved; T.TIME: The total CPU time for 3,042 (both buggy and clean) RealWorld programs

	C.SOLV	T.TIME
LA	1449	772, 501s
LA.T.st	1, 500 (↑ **4**%)	822, 982s (↑ 7%)
LA.T.bb	1, 500 (↑ **4**%)	817, 056s (↑ 6%)
LA.T.lh	1, 471 (↑ **2**%)	730, 531s (↓ **5**%)
LA.T.lf	1, 499 (↑ **3**%)	827, 401s (↑ 7%)

5.2.4 RQ4: Effectiveness of the distance metric selection

The results show that TOUR with *program-specific distance metric selection outperforms* TOUR with a single-distance metric. BAM.T solves from 20 (= 2,116-2,096) to 193 (= 2,116-1,923) more programs by taking from 2% (= (732,531-717,901)/732,531) to 12% (= (924,237-717,901)/924,237) less total CPU time than BAM.T.{st,bb,lh,lf} techniques as shown in Table 5.10. LA.T solves from 47 (= 1,518-1,471) to 18 (= 1,518-1,500) more programs than LA.T.{st,bb,lh,lf} techniques as shown in Table 5.11. LA.T takes from 3% (= (827,401-799,777)/827,401) to 2% (= (817,056-799,777)/817,056) less total CPU-time than LA.T.{st,bb,lf} while LA.T.lh takes less total CPU-time than LA.T.

Note that the model selection technique that I use needs historical results. Thus, it is impossible to use the selection method if sufficient historical results are not available. I discuss the number of necessary historical results for reasonable selection model generation at Section 5.2.5.

5.2.5 RQ5: Effectiveness of the historical result filtering

The results show that the historical result filtering is effective to generate a better selection model, and the threshold of 40 seconds is the best for both BAM.T and LA.T. BAM.T.40 solves 2% more programs in 3% less total CPU time than than BAM.T.NO for 3,042 RealWorld programs. LA.T.40 solves 3% more programs in 10% more total CPU time than LA.T.NO for 3,042 RealWorld programs. For BAM.T, all threshold values are effective to generate a better selection model compared to the no filtering. For LA.T all threshold values are effective to generate a selection model that solves more programs, but their generated models make TOUR take more time. Even with a small threshold value

Table 5.10: The number of solved programs and total CPU time of BAM.T and BAM.T.{st,bb,lh,lf}; C.SOLV: the number of programs solved; T.TIME: The total CPU time for 3,042 (both buggy and clean) RealWorld programs

	C.SOLV	T.TIME
BAM.T	2, 116	717, 901s
BAM.T.st	1, 923 (↓ **10**%)	924, 237s (↑ **12**%)
BAM.T.bb	2, 044 (↓ **4**%)	791, 910s (↑ **9**%)
BAM.T.lh	2, 030 (↓ **4**%)	789, 454s (↑ **9**%)
BAM.T.lf	2, 096 (↓ **1**%)	732, 531s (↑ **2**%)

Table 5.11: The number of solved programs and total CPU time of LA.T.{st,bb,lh,lf}; C.SOLV: the number of programs solved; T.TIME: The total CPU time for 3,042 (both buggy and clean) RealWorld programs

	C.SOLV		T.TIME	
LA.T	1,518		799,777s	
LA.T.st	1,500	(↓ **1**%)	822,982s	(↑ **3**%)
LA.T.bb	1,500	(↓ **1**%)	817,056s	(↑ **2**%)
LA.T.lh	1,471	(↓ **3**%)	730,531s	(↓ 10%)
LA.T.lf	1,499	(↓ **1**%)	827,401s	(↑ **3**%)

Table 5.12: The number of solved programs and total CPU time of BAM.T.{10,20,30,40,50}; C.SOLV: the number of programs solved by BAM.T.{10,20,30,40,50} (difference ratio compared to that of BAM.T.NO that solves 2,081 programs); T.TIME: The total CPU time of BAM.T.{10,20,30,40,50} for 3,042 RealWorld programs (difference ratio compared to that of BAM.T.NO that takes 737,983 seconds of CPU time); N.P: the number of historical results used for the selection model generation

BAM.T.NO vs.	C.SOLV		T.TIME		N.P
BAM.T.10	2,106	(↑ 1%)	723,643s	(↓ 2%)	721
BAM.T.20	2,111	(↑ 1%)	720,534s	(↓ 2%)	608
BAM.T.30	2,116	(↑ 2%)	720,427s	(↓ 2%)	529
BAM.T.40	**2,116**	(↑ **2**%)	**717,705s**	(↓ **3**%)	500
BAM.T.50	2,113	(↑ 2%)	718,190s	(↓ 3%)	489

(i.e., 10 seconds), the number of programs is significantly reduced compared to the no filtering (BAM.T.10 filters out 76% of programs, i.e., 2,321 out of 3042, and LA.T.10 filters out 94% of programs, i.e., 2,851 out of 3042). The decreased number of reduced historical results as increasing threshold is not significant in both BAM.T and LA.T. There is no relationship between increasing threshold and the effectiveness of the filtering.

The results show that the necessary number of historical results for effective model generation is relatively small on both BAM.T and LA.T. However, it may not easy to collect historical results that satisfy the filtering threshold. Since there are many benchmark programs and open source programs for verification, it is more practical to collect many program in different domains to generate a model and then to apply the historical result filtering. Note that still TOUR using a single distance metric is effective for speeding up ARMC.

5.3 Discussion

5.3.1 Programs that TOUR is Bad

The two main reasons that TOUR is worse than the existing ARMC are (1) a bad distance metric for reducing the number of constructed states and (2) an iterative loop exploration which induces much exploration time despite the less number of constructed states. Figure 5.4 shows the number of reduced constructed states and the time saving between BAM and BAM.T.st (left), and BAM and BAM.T.lf

Table 5.13: The number of solved programs and total CPU time of LA.T.{10,20,30,40,50}; C.SOLV: the number of programs solved by LA.T.{10,20,30,40,50} (difference ratio compared to that of LA.T.NO that solves 1,481 programs); T.TIME: The total CPU time of LA.T.{10,20,30,40,50} for 3,042 RealWorld programs (difference ratio compared to that of LA.T.NO that takes 729,926 seconds of CPU time); N.P: the number of programs used for the selection model generation

LA.T.NO vs.	C.SOLV		T.TIME		N.P
LA.T.10	$1,516$	($\uparrow$ 2%)	$807,931$s	($\uparrow$ 11%)	191
LA.T.20	$1,514$	($\uparrow$ 2%)	$791,991$s	($\uparrow$ 9%)	154
LA.T.30	$1,513$	($\uparrow$ 2%)	$810,448$s	($\uparrow$ 11%)	138
LA.T.40	$\mathbf{1,518}$	($\uparrow$ **3**%)	$799,639s$	($\uparrow$ 10%)	131
LA.T.50	$1,517$	($\uparrow$ 2%)	$806,050$s	($\uparrow$ 10%)	126

(right). The left graph shows the 1,825 RealWorld programs that both BAM and BAM.T.st solve. The right graph shows the 1,978 RealWorld programs that both BAM and BAM.T.lf solve. The x-axis denotes the constructed states difference between BAM and BAM.T.st (BAM.T.lf) in log scale and the y-axis denotes the CPU time difference between BAM and BAM.T.st (BAM.T.lf) in log scale. The quadrant 1 denotes BAM.T.st (BAM.T.lf) takes less CPU time and constructs less states than BAM. The quadrant 2 denotes BAM.T.st (BAM.T.lf) takes less CPU time and constructs more states than BAM. The quadrant 3 denotes BAM.T.st (BAM.T.lf) takes more CPU time and constructs more states than BAM. The quadrant 4 denotes BAm.T.st (BAM.T.lf) takes more CPU time and constructs less states than BAM.

The programs in quadrant 3 show that TOUR takes more CPU time with more constructed states than BAM. To save the time of the programs in quadrant 3, it is necessary to apply another search heuristic that reduces the constructed states. The figure shows that the number of programs in quadrant 3 is reduced by applying lf as the distance metric compared to that of using st as the distance metric (from 201 to 108 programs). The lf metric reduces the constructed states more compared to the st metric because lf calculates the distance based on the number of abstract states that BAM constructs to reach an error location. BAM utilizes block encoding [13, 12] and constructs an abstract state for a loop head or function entry/exit location that lf assigns an weight.

The programs in quadrant 4 show that TOUR takes more CPU time than BAM while TOUR constructs less states than BAM, which is different than I expected. For the programs in quadrant 4, TOUR explores loops iteratively, which induces much exploration time despite the less constructed states. I show the number of coverage-check indicating how a technique iteratively explores the same locations (i.e., iterative loop exploration) with time consuming operations [59]. [2] Figure 5.5 shows the constructed states difference and the coverage-check difference between BAM and BAM.T.st (left) and BAM and BAM.T.lf (right). The intuition is that the coverage-check difference is proportional to the constructed states difference because more constructed states may induce more coverage-check. The programs in quadrant 4 show that TOUR constructs less states than BAM while TOUR conducts more coverage-checks than BAM, which is different than the above intuition.

Using an appropriate distance metric is also effective to avoid the iterative loop exploration. Fig-

[2] An ARMC technique checks the coverage of an abstract state $\langle l, a\rangle$ with other abstract states corresponding to the same location l (i.e., $\langle l, a'\rangle$). If $\exists a'.a \sqsubseteq a'$, $\langle l, a\rangle$ is covered by $\langle l, a'\rangle$ and an ARMC technique stops state exploration from a covered abstract state.

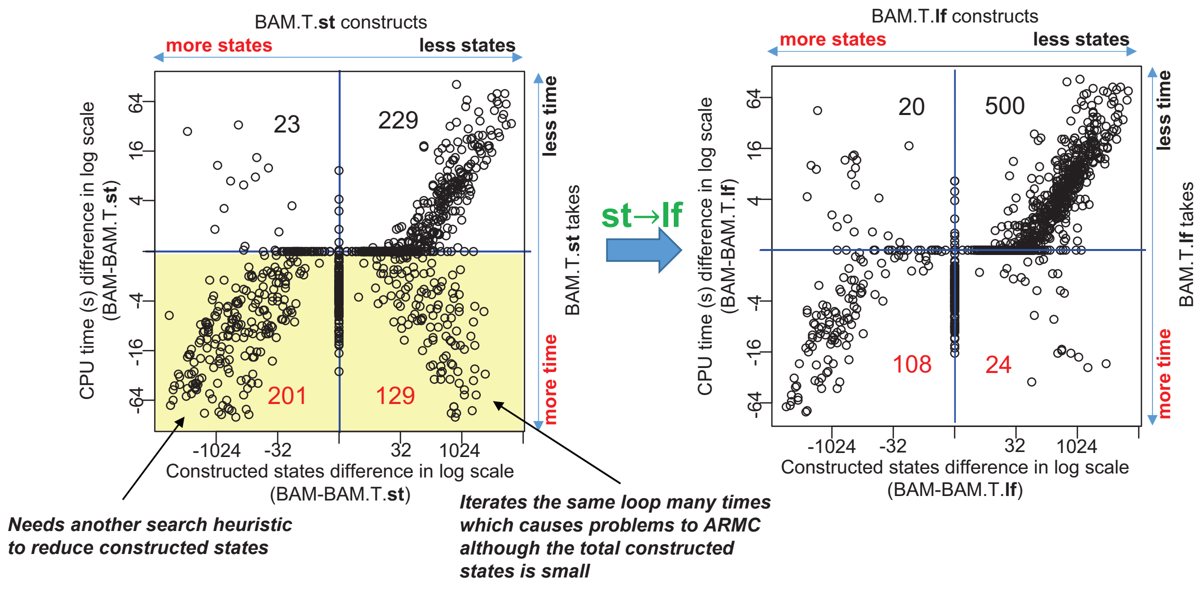


Figure 5.4: (Left) For 1,825 RealWorld programs that both BAM and BAM.T.st solve, the CPU time difference between BAM and BAM.T.st in log scale for y-axis and the constructed states difference between BAM and BAM.T.st in log scale for x-axis; (Right) For 1,978 RealWorld programs that both BAM and BAM.T.lf solve, the CPU time difference between BAM and BAM.T.lf in log scale for y-axis and the constructed states difference between BAM and BAM.T.lf in log scale for x-axis

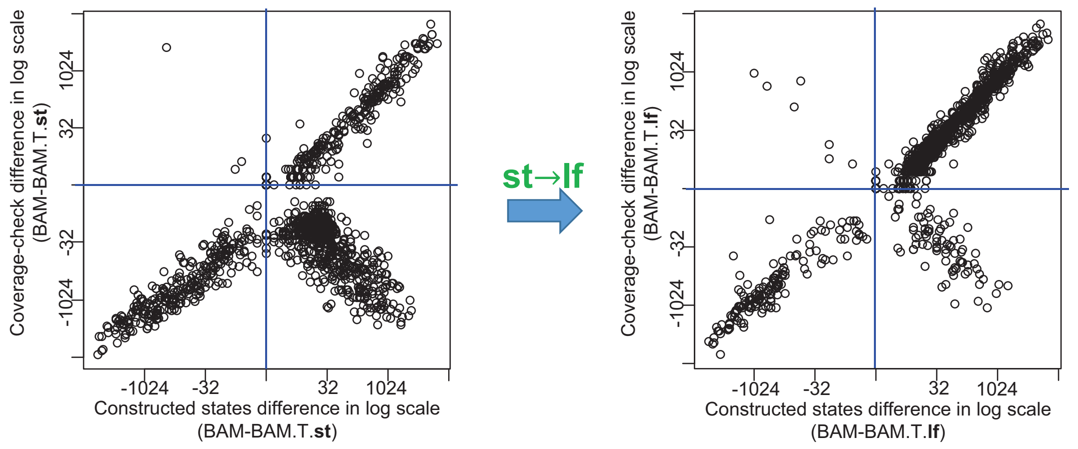


Figure 5.5: (Left) For 1,825 RealWorld programs that both BAM and BAM.T.st solve, the number of coverage-check difference between BAM and BAM.T.st in log scale for y-axis and the constructed states difference between BAM and BAM.T.st in log scale for x-axis; (Right) For 1,978 RealWorld programs that both BAM and BAM.T.lf solve, the number of coverage-check difference between BAM and BAM.T.lf in log scale for y-axis and the constructed states difference between BAM and BAM.T.lf in log scale for x-axis

Table 5.14: The number of programs in SOLV or NOT-SOLV for each project of RealWorld

	Total	SOLV	NOT-SOLV
LDV	2,728	2,142	461
AWS	176	18	39
UTHASH	138	0	96

ure 5.5 shows the constructed states difference and the coverage-check difference between BAM and BAM.T.st (left) and BAM and BAM.T.lf (right). By using the distance metric lf instead of the distance metric st, the number of programs in the quadrant 4 in Figure 5.5 is significantly reduced.

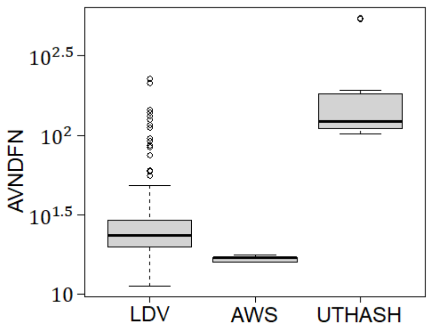
5.3.2 Programs that ARMC cannot Solve

I investigate which programs are hard-to-solve by ARMC-based techniques to give a practical guide for ARMC application. I identify RealWorld programs solved by any of BAM and BAM.T.{st,bb,lh,lf} as SOLV and failed by all of BAM and BAM.T.{st,bb,lh,lf} due to time-out as NOT-SOLV. I exclude the programs failed due to reasons other than time-out because the failures are caused by the limitation of the current ARMC-based techniques' implementation. Then I compare the static program feature distribution of SOLV and NOT-SOLV. Since 3,042 RealWorld programs are collected from three different real-world open source projects (i.e., 2,728 programs from Linux Driver Verification (LDV), 178 programs from aws C common library (AWS), and 138 programs from uthash (UTHASH)), I show the distribution of programs for each project individually.

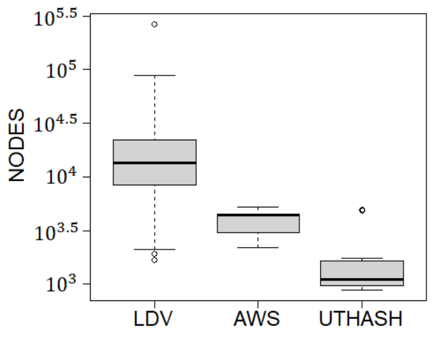
The ARMC-based techniques are better to solve LDV programs than AWS or UTHASH programs. Table 5.14 shows the number of programs in SOLV or NOT-SOLV for each project. The ARMC-based techniques solve 2,142 LDV programs, 18 AWS programs, and 0 UTHASH programs. They time out for 461 LDV programs, 39 AWS programs, and 96 UTHASH programs. One analysis show that the bigger average size of functions make UTHASH programs hard-to-solve. Figure 5.6a shows the AVNDFN value (i.e., the static feature value representing the average number of statements of functions) distribution of NOT-SOLV programs in each project of RealWorld. The UTHASH programs have much bigger AVNDFN than LDV or AWS programs, which mean each function of a UTHASH program has too many statements to analyze by ARMC-based techniques. The total program size is not matter about the difference between projects. Figure 5.6b shows the NODES value (i.e., the static feature value representing the number of statements) distribution of NOT-SOLV programs in each project of RealWorld. Although the AWS and UTHASH programs are much smaller than LDV programs, ARMC-based techniques better to solve LDV programs. Specifically, UTHASH programs are significantly smaller than LDV programs, but ARMC-based techniques cannot solve any UTHASH programs.

There are two things that the further investigations are necessary. First, the NODES and AVNDFN distribution of NOT-SOLV LDV programs are widespread, which means there are more reasons that the programs are not solvable by ARMC-based techniques. Second, the NODES and AVNDFN values of NOT-SOLV AWS programs are relatively small compared to LDV and UTHASH while ARMC-based techniques are bad to solve AWS programs. It is necessary to investigate why AWS programs are not solvable by ARMC-based techniques.

I also investigate the difference of distribution between SOLV and NOT-SOLV programs for each static program features. For LDV programs, program size is the most relevant factor to decide the



(a)



(b)

Figure 5.6: (a) The AVNDFN value distribution of NOT-SOLV programs in each project of RealWorld; (b) The NODES value distribution of NOT-SOLV programs in each project of RealWorld

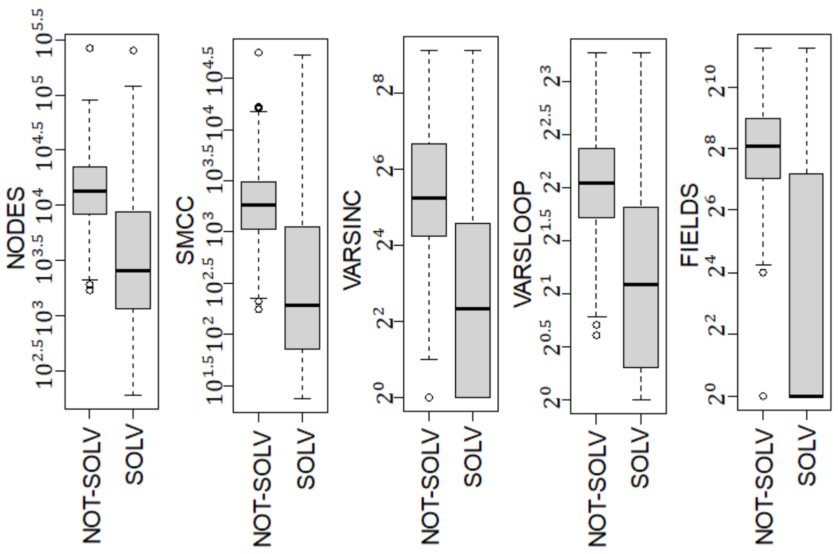


Figure 5.7: The NODES, SMCC, VARSINC, VARSLOOP, and FIELDS value distribution of NOT-SOLV and SOLV programs in LDV of RealWorld

solvability of ARMC-based techniques. Figure 5.7 shows the distribution of NODES, SMCC, VARSINC, VARSLOOP, and FIELDS values of NOT-SOLV and SOLV programs in LDV of RealWorld. All the feature values are increasing as the increasing size of programs. For the five features, the SOLV programs have smaller values than the NOT-SOLV programs. Although the size is mainly affect the solvability of ARMC-based techniques, a further investigation about other reasons of the solvability for bigger-sized SOLV programs.

The effect of program size to the solvability of ARMC-based techniques is not shown in AWS programs. Figure 5.8 shows the NODES value distribution of NOT-SOLV and SOLV programs in AWS of RealWorld. In AWS, the distribution of NODES values between NOT-SOLV and SOLV programs are not big to distinguish each other. The distribution of NODES values is relatively narrow for both NOT-SOLV and SOLV. Unfortunately, I do not find program features to distinguish NOT-SOLV and SOLV from the 25 static program features.

I do not analyze the distribution analysis for UTHASH because the UTHASH does not have SOLV

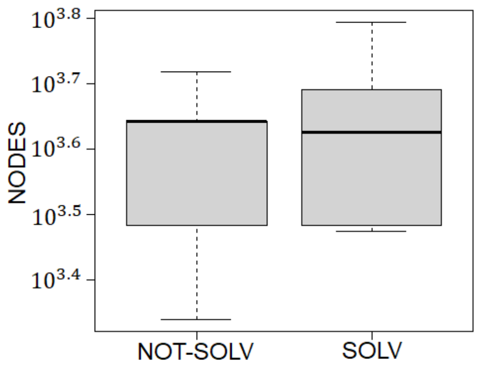


Figure 5.8: The NODES value distribution of NOT-SOLV and SOLV programs in AWS of RealWorld programs.

Chapter 6. Conclusion

In this paper, I introduced TOUR which improves both the bug detection ability and the verification speed of ARMC. TOUR is the first ARMC technique that applies directed search (based on the distance to an error location) for fast target path detection in interprocedural programs. Additionally, as shown in Section 5.2.4, the program-specific distance metric selection heuristic of TOUR contributes to improving the performance of ARMC. The experimental results on the 3,042 real-world C program benchmark confirm the improved model checking performance of TOUR. For example, LA using TOUR finds bugs in 118% more programs than LA for 354 real-world buggy C programs, and BAM using TOUR finds bugs in 20% more programs than BAM in 11% less total CPU time than BAM. BAM using TOUR verifies 15% more programs in 15% less total CPU time for 652 real-world complex clean C programs. In conclusion, TOUR achieves fast reachability analysis by speeding up ARMC, and the fast reachability analysis helps practitioners to efficient software reliability assurance.

6.1 Future Research Direction

Developing a distance metric using not only program syntax, but also program semantic obtained at runtime is a research direction. As I analyzed, the static program features representing the program syntax are insufficient to analyze the ARMC's weakness. The program semantics are one candidate to analyze the ARMC's weakness further. By analysing why ARMC-based techniques are fails to verify a small-sized programs using program semantics, someone can develop a new distance metric to mitigate the ARMC's weakness and speed up ARMC further.

Developing a distance metric for verification properties other than reachability (e.g., termination property) is another research direction. Guiding an ARMC to find a target path with infinite loop (i.e., a target path for a termination property) is an example. If a target path with infinite loop is detected and it is spurious, ARMC may update the abstraction to eliminate the spurious infinite loop path. The distance metric for exploring loops with high priority may not effective because loop explorations are time consuming in ARMC as I analyze. Which metrics are highly related to the infinite loop exploration is the key technical challenge of this research direction.

Handling multiple error locations efficiently is a research direction. In real-world programs, there are many error locations in a program. While I use the same weight for the multiple error locations, it is possible to assign different weights for different error locations. The different weight allows to focus a specific error location instead of considering multiple error locations simultaneously without focusing. Also, the weight may consider which error location actually has an actual bug path using program semantics.

Chapter 7. Appendix

7.1 An Effective Implementation of the Directed ARMC of TOUR for Block-Abstraction Memoization

The directed ARMC algorithm of TOUR can hinder the efficiency of Block-Abstraction Memoization for interprocedural program analysis (BAM), which is a state-of-the-art ARMC technique caching a sub-state-graph of a function to reuse [14]. The distance values of the states in a cached sub-state-graph might be inconsistent with the function call context of when the cached graph is reused because the function call abstract states of the sub graph are changed.

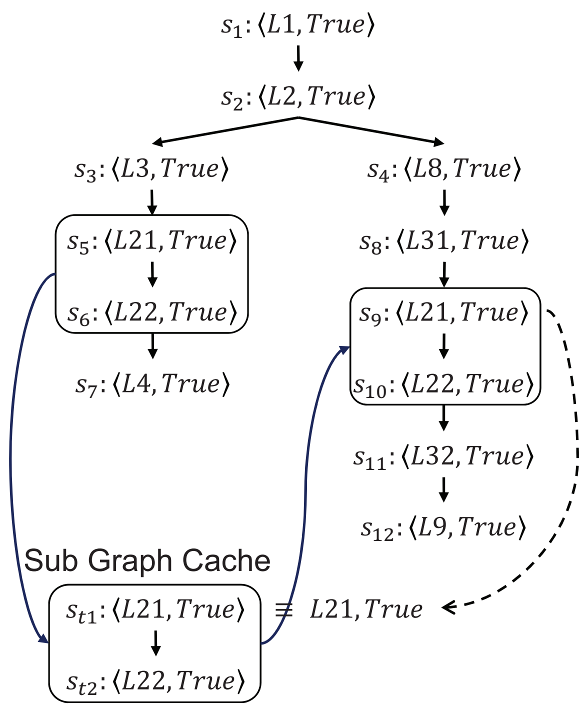
Figure 7.2 shows an intermediate abstract state graph constructed by the directed ARMC with BAM for the example program in Figure 7.1 which is modified from the program in Figure 3.1 by invoking the function `g` at `L8` and the function `g` invokes `f`. The graph is constructed in an arbitrary order of visiting abstract states. The states s_5 and s_6 constitute a sub-state-graph of the function `f` and BAM caches the sub-state-graph. When the algorithm visits the abstract state s_8, its successor abstract state corresponds to the location `L21` and has the abstract data state as $True$. The successor abstract state is same as the cached abstract state s_{t1}, and the algorithm thus gets the cached sub-state-graph as abstract states s_8 and s_9 instead of explicitly constructing them. By the way, a problem occurs at s_{11}, the successor abstract state of the reused sub-state-graph. Since the function call abstract state of the reused sub-state-graph is s_3, the s_{11}'s function call abstract state is nil (=`call`(`call`(s_{10}))) as shown in the `map1` of Figure 7.2. However, the correct function call abstract state of s_{11} is s_4. Due to the incorrect function call context of the reused sub-state-graph, the distance values since s_9 are totally incorrect. Besides, the algorithm cannot obtain s_{12}'s function call abstract state because s_{11}'s function call abstract state is nil. One solution to solve the problem is to update the function call abstract state of all abstract states in the reused sub-state-graph, but it hinders the benefit of the cache and reuse mechanism.

To avoid the problem I extend the directed ARMC algorithm of TOUR to minimize the function call abstract state updating. The extended algorithm does not store the function call context at all abstract states. The algorithm instead stores the function call context only at function entry abstract states (e.g., s_1, s_5, s_8, and s_9 in Figure 7.2). And it maintains the mapping function from an abstract state to the function entry abstract state. Since the function call context is stored only in function entry abstract states, the extended algorithm reduces the number of function call context updates when reusing a sub-state-graph compared to the original algorithm.

Figure 7.3 shows the extended directed ARMC algorithm of TOUR. The shaded lines are differences between the extended algorithm and the original algorithm in Figure 3.13. The extended algorithm calculates the function entry abstract state of each abstract state. If *op* is a function call operation, the function entry abstract state of $\langle l', a' \rangle$ (i.e., the successor abstract state of the selected abstract state) is itself (line 9). If *op* is a function return operation, the function entry abstract state of $\langle l', a' \rangle$ is the function entry abstract state of the $\langle l, a \rangle$'s function call abstract state (line 11). Since only function entry abstract states have the function call context, the algorithm obtains the function entry abstract state of $\langle l, a \rangle$ to get the function call abstract state (line 11). In contrast to the original algorithm, the extended algorithm does not compute the function call abstract state of $\langle l', a' \rangle$ in the function return

```
     void main(int *x,int k){
 L1:   int i;
 L2:   if(x[1]>x[2]){
 L3:     f(x[1]);
 L4:     if(k==0){
 L5:       x[2]=x[1];
         }else{
 L6:       x[1]=x[2];
 L7:     }
       }else{
 L8:     g(x[2]);
 L9:   }
L10:   if(x[1]>x[2]){
L11:     abort();
       }
L12:}
     void f(int i){
L21:   int a=i;
L22:}
     void g(int i){
L31:   f(i);
L32:}
```

Figure 7.1: An example C program



map1

states	dist	call
s_1	5	nil
s_2	4	nil
s_3	5	nil
s_4	3	nil
s_5	5	s_3
s_6	4	s_3
s_{t1}	5	s_3
s_{t2}	4	s_3
s_7	4	nil
s_8	3	s_3
s_9	**5**	$\boldsymbol{s_3}$
s_{10}	**4**	$\boldsymbol{s_3}$
s_{11}	**4**	nil
s_{12}	**4**	?

map2

states	dist	call	entry
s_1	5	nil	s_1
s_2	4	–	s_1
s_3	5	–	s_1
s_4	3	–	s_1
s_5	5	s_3	s_5
s_6	4	–	s_5
s_{t1}	5	–	s_{t1}
s_{t2}	4	–	s_{t1}
s_7	4	–	s_1
s_8	3	s_4	s_8
s_9	3	s_8	s_9
s_{10}	2	–	s_9
s_{11}	2	–	s_8
s_{12}	2	–	nil

Figure 7.2: An example abstract state graph constructed by BAM for the example program in Figure **??**, which shows the problem of applying the directed ARMC of TOUR to BAM

operation case because $\langle l', a' \rangle$ is not a function entry. If *op* is not a function call or return operation, the function entry abstract state of $\langle l', a' \rangle$ is the same as the function entry abstract state of $\langle l, a \rangle$ (line 13). The algorithm does not compute the function call abstract state of $\langle l', a' \rangle$ at this cases because $\langle l', a' \rangle$ is not a function entry. For the distance calculation, the algorithm uses the function entry abstract state of $\langle l', a' \rangle$ to get the function call context (line 15).

The map2 of Figure 7.2 shows the values of mapping functions when the extended algorithm is used for the abstract state graph construction. The algorithm can update the function call abstract state of the two abstract states (i.e., s_9 and s_{10}) in the reused state graph by updating the function call abstract state of the function entry s_9. Since the function call abstract states are consistent, successor states are also have consistent function call abstract states and the distance values.

Since the distance values are frequently changed due to the cache and reuse mechanism, the extended algorithm does not compute the distance at the construction time. The algorithm instead computes the distance of an abstract state whenever the distance is actually needed. Figure 7.4 shows the implementation of the mapping function `dist` for the on-demand distance calculation. I utilize the distance value caching (i.e., CD) for the mapping function `dist` to reduce the number of recursive `dist` calls. The algorithm takes s, the abstract state to calculate the distance, and uses a global variable CD, the mapping function from a function call abstract state to the integer indicating the distance of the state. If CD does not have the distance value of the function call abstract state of s (i.e., `call`(`entry`(s))), the algorithm compute the distance of `call`(`entry`(s)) and save the distance value to CD (line 1–3). The algorithm calculates the distance of s using the cached distance value of `call`(`entry`(s)) in CD and the exit-dist and abs-dist of the corresponding location l (line 4).

7.2 Calculating Program Features

I demonstrate how to calculate each program feature in Table 4.1 from a C program in detail using the example program in Figure 7.5 and the corresponding control-flow graph in Figure 7.6:

- **LOCS:** The feature LOCS means the number of locations in a program. I calculate LOCS as the number of control-flow nodes constructed by Eclipse CDT integrated in the open source model checking tool CPAchecker [63]. The CPAchecker utilizes a lot of intermediate variables to make one location has one operation. Thus, LOCS is much more bigger than the number of lines on the C file. The LOCS value of the example program in the example program is 17.

- **MXLOFN:** The feature MXLOFN means the maximum number of locations among the functions in a program. It is calculated as the maximum of LOCS values of each function in a program. The LOCS value of a function is the number of locations between the function entry location and the function exit location. The LOCS value of the function main is 12 and that of the function f is 5 in the example program. Thus, the MXLOFN value of the example program is 12.

- **AVLOFN:** The feature AVLOFN means the average number of locations among the functions in a program. It is calculated as the average of LOCS values of each function in a program. The AVLOFN value of the example program is 8.5 (=(12+5)/2).

- **SDLOFN:** The feature SDLOFN means the standard deviation of the number of locations among the functions in a program. It is calculated as the standard deviation of LOCS values of each function in a program. The SDLOFN value of the example program is 3.5 ($=\sqrt{((12-8.5)^2+(5-8.5)^2)/2}$).

Input: a set of abstract state graph nodes N; a set of abstract state graph edges E; a set of error locations L_{err}; **Global Variables:** `call`, a mapping function from an abstract state to the function call abstract state; `dist`, a mapping function from an abstract state to the distance of the state; `entry`, a mapping function from an abstract state to the function entry abstract state of the state

Output: $safe$ if the graph contains no target path or $\langle unsafe, N, E\rangle$ if the graph contains a target path

```
1:  reached := $N$
2:  waitlist := unmarked nodes from $N$
3:  while waitlist$\neq \emptyset$ do
4:    $\langle l, a\rangle$ := pop an abstract state with the shortest distance from waitlist
5:    for all outgoing operations op of $l$ do
6:      $\langle l', a'\rangle := \langle l, a\rangle$'s successor abstract state according to op
7:      if op is a function call operation then
8:        call($\langle l', a'\rangle$) := $\langle l, a\rangle$
9:        entry($\langle l', a'\rangle$) := $\langle l', a'\rangle$
10:     else if op is a function return operation then
11:       entry($\langle l', a'\rangle$) := entry(call(entry($\langle l, a\rangle$)))
12:     else
13:       entry($\langle l', a'\rangle$) := entry($\langle l, a\rangle$)
14:     end if
15:     dist($\langle l', a'\rangle$) := min($l'$.exit-dist + dist(call(entry($\langle l', a'\rangle$))), $l'$.abs-dist)
16:     reached := reached $\cup \{\langle l', a'\rangle\}$
17:     $E := E \cup \{\langle l, a\rangle \rightarrow \langle l', a'\rangle\}$
18:     if $a' \neq False$ then
19:       if $l' \in L_{err}$ then
20:         return $\langle unsafe$, reached, $E\rangle$
21:       end if
22:       waitlist := waitlist $\cup \{\langle l', a'\rangle\}$
23:     end if
24:   end for
25:   mark $\langle l, a\rangle$
26: end while
27: return $safe$
```

Figure 7.3: The directed ARMC algorithm of TOUR extended with the mapping function `entry`

Input: $s = \langle l, a\rangle$, an abstract state to calculate the distance; **Global Variables:** `call`; `entry`; and CD, the mapping function from a function call abstract state to the integer indicating the distance of the state

Output: the distance of the state s

```
1: if $CD$ does not have the call(entry($s$))' value then
2:   $CD$(call(entry($s$))) := min($l$.exit-dist + dist(call(entry($s$))), $l$.abs-dist)
3: end if
4: return min($l$.exit-dist + $CD$(call(entry($s$))), $l$.abs-dist)
```

Figure 7.4: Algorithm: `dist`

- **EDGES:** The feature EDGES means the number of operations in a program. I calculate EDGES as the number of control-flow edges constructed by Eclipse CDT integrated in CPAchecker. The EDGES value of the example program is 21 including function call and return operation edges.

- **MXEGFN:** The feature MXEGFN means the maximum number of control-flow edges among the functions in a program. It is calculated as the maximum of EDGES values of each function in a program. The EDGES value of a function is the number of control-flow edges between the function entry location and the function exit location. The EDGES value of the function main is 13 including implicit edges between a function call location and a function return location (e.g., L3 $\rightarrow$ L4) and that of the function f is 6 in the example program. Thus, the MXEGFN value of the example program is 13.

- **AVEGFN:** The feature AVEGFN means the average number of control-flow edges among the functions in a program. It is calculated as the average of EDGES values of each function in a program. The AVEGFN value of the example program is 9.5 (=(13+6)/2).

- **SDEGFN:** The feature SDEGFN means the standard deviation of the number of control-flow edges among the function in a program. It is calculated as the standard deviation of EDGES values of each function in a program. The SDEGFN value of the example program is 3.5 ($=\sqrt{((13-9.5)^2+(6-9.5)^2)/2}$).

- **LOOPS:** The feature LOOPS means the number of loops in a program. It is calculated as the number of loop head locations in a program. I utilizes the loop structure construction algorithm of CPAchecker to find loop head locations. The LOOPS value of the example program is 2 (i.e., L22 and L23).

- **MXLPFN:** The feature MXLPFN means the maximum number of loop head locations among the functions in a program. It is calculated as the maximum of LOOPS values of each function in a program. The LOOPS values of the function main is 0 and that of the function f is 2 in the example program Thus, the MXLPFN value of the example program is 2.

- **AVLPFN:** The feature AVLPFN means the average number of loop head locations among the functions in a program. It is calculated as the average of LOOPS values of each function in a program. The AVLPFN value of the example program is 1 (=(0+2)/2).

- **SDLPFN:** The feature SDLPFN means the standard deviation of the number of loop head locations among the functions in a program. It is calculated as the standard deviation of the LOOPS values of each function in a program. The SDLPFN value of the example program is 1 ($=\sqrt{((0-1)^2+(2-1)^2)/2}$).

- **FUNCS:** The feature FUNCS means the number of functions in a program. The FUNCS value of the example program is 2 (i.e., the function main and the function f).

- **MXCALLS:** The feature MXCALLS means the maximum number of function calls per function in a program. It is calculated as the maximum of the CALLS values of each function in a program, where CALLS of a function means the number of function call locations of the function. The CALLS value of the function main is 0 and that of the function f is two. Thus, the MXCALLS value of the example program is 2 (i.e., L3 and L8).

- **AVCALLS:** The feature AVCALLS means the average number of function calls per function in a program. It is calculated as the average of the CALLS values of each function in program. The AVCALLS value of the example program is 1 (=(0+2)/2).
- **SDCALLS:** The feature SDCALLS means the standard deviation of function calls per function in a program. It is calculated as the standard deviation of the CALLS values of each function in a program. The SDCALLS value of the example program is 1 ($=\sqrt{((0-1)^2+(2-1)^2)/2}$).
- **VARS:** The feature VARS means the number of variables used in branch conditions and relevant to the branch condition variables in a program. It also includes the variables used for pointer dereferencing. I utilize the variable classification module of CPAchecker. It first finds the branch condition variables and propagates with regard to the dependency of variables. The VARS value of the example program is 5 (i.e., main:x, main:k, f:i, f:a, and f:b).
- **VARSASM:** The feature VARSASM means the number of variables that are used in branch conditions in a program. The VARSASM value of the example program is 5 (i.e., main:x, main:k, f:i, f:a, and f:b).
- **VARSLOOP:** The feature VARSLOOP means the number of variables that are used in loop terminate conditions in a program. The VARSLOOP value of the example program is 3 (i.e., f:i, f:a, and f:b).
- **VARSINC:** The feature VARSINC means the number of variables that are used in for statements as the increasing/decreasing counter variable in a program. The VARSINC value of the example program is 2 (i.e., f:a and f:b).
- **FIELDS:** The feature FIELDS means the number of bit field variables used in branch conditions and relevant to the branch condition variables in a program. The FIELDS value of the example program is 0 (no bit field variable).
- **MXCC:** The feature MXCC means the maximum cyclomatic complexity per function of a program. Since target programs are complex and interprocedural with recursive function calls, it is hard to compute cyclomatic complexity of whole program. Instead I compute the cyclomatic complexity of each function of the program and the representative values of the cyclomatic complexity of functions. The cyclomatic complexity value of a function (CC) is calculated as $\text{EDGES} - \text{LOCS} + 2$ of the function because a function as one function entry location and one function exit location. The CC of the function main is 3 (=13-12+2) and that of the function f is 3 (=6-5+2). Thus, the MXCC value of the example program is 3.
- **SMCC:** The feature SMCC means the sum of cyclomatic complexity per function of a program. It is calculated as the sum of CC values of each function of the program. The SMCC value of the example program is 6 (=3+3).
- **AVCC:** The feature AVCC means the average cyclomatic complexity of functions in a program. It is calculated as the average of CC values of each function in the program. The AVCC value of the example program is 3 (=(3+3)/2).
- **SDCC:** The feature SDCC means the standard deviation cyclomatic complexity of functions in a program. It is calculated as the standard deviation of CC values of each function of the program. The SDCC value of the example program is 0 ($=\sqrt{((3-3)^2+(3-3)^2)/2}$).

```
    void main(int *x,int k){
 L1:   int i;
 L2:   if(x[1]>x[2]){
 L3:      f(x[1]);
 L4:      if(k==0){
 L5:         x[2]=x[1];
          }else{
 L6:         x[1]=x[2];
 L7:      }
       }else{
 L8:      f(x[2]);
 L9:   }
L10:   if(x[1]>x[2]){
L11:      abort();
       }
L12:}
    void f(int i){
L21:   int a,b,c;
L22:   for(a=0;a<i;a++){
L23:      for(b=0;b<i;b++){
L24:         c++;
          }
       }
L25:}
```

Figure 7.5: An example C program

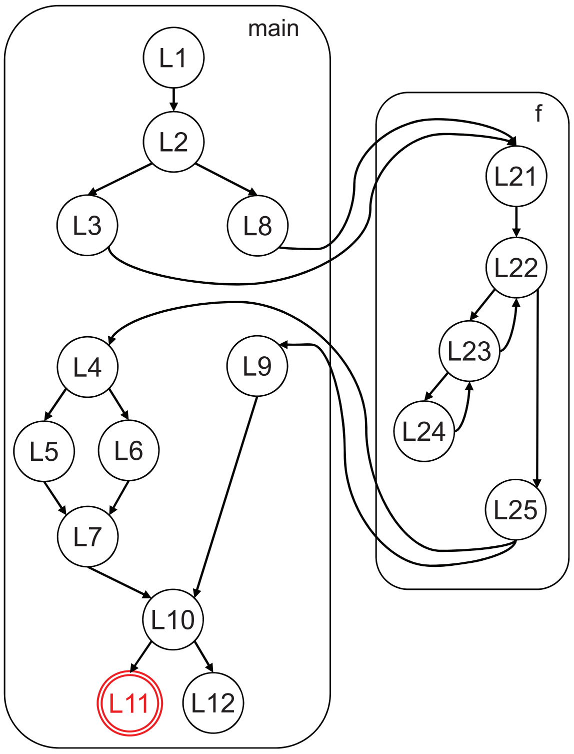


Figure 7.6: The control-flow graph of the example program in Figure 7.5

7.3 The Extended Program Annotation Algorithms

```
Input: $f$, the function name to compute exit-dist; Global Variables: $FW$, the mapping function from
    function name to the function's shortest distance; $W$, the mapping function from a control-flow edge
    to integer indicating the edge weight
 1: $l_{exit}$ := the function exit location of $f$
 2: $l_{entry}$ := the function entry location of $f$
 3: $l_{exit}$.exit-dist:= 0
 4: $VS := \emptyset$
 5: $Q := \emptyset$
 6: $VS := VS \cup \{l_{exit}\}$
 7: $Q := Q \cup \{l_{exit}\}$
 8: while $Q \neq \emptyset$ do
 9:   $l$ := pick a location with the smallest exit-dist from $Q$
10:   for all incoming operations $e$ of $l$ do
11:     $p$ := $e$'s source location
12:     if $p \in VS$) then
13:       continue
14:     end if
15:     if $e$ is not a function return operation then
16:       $p$.exit-dist:= $W(e) + l$.exit-dist
17:     else
18:       $p$ := the corresponding function call location of $l$
19:       $p$.exit-dist:= $FW(p'$s function name$) + l$.exit-dist
20:     end if
21:     $VS := VS \cup \{p\}$
22:     if $p \neq l_{entry}$ then
23:       $Q := Q \cup \{p\}$
24:     end if
25:   end for
26: end while
27: $FW(f) := l_{entry}$.exit-dist
```

Figure 7.7: Algorithm: `exit-dist`

Input: L_{err}, the set of error locations; **Global Variables:** FW, the mapping function from function name to the function's shortest distance; W, the mapping function from a control-flow edge to integer indicating the edge weight

```
1: $VS := \emptyset$
2: $Q := \emptyset$
3: for all $l_{err} \in L_{err}$ do
4:     $l_{err}$.abs-dist:= $0$
5:     $VS := VS \cup \{l_{err}\}$
6:     $Q := Q \cup \{l_{err}\}$
7: end for
8: while $Q \neq \emptyset$ do
9:     $l$ := pick a location with the smallest abs-dist from $Q$
10:    for all incoming operations $e$ of $l$ do
11:        $p := e$'s source location
12:        if $p \in VS$ then
13:            continue
14:        end if
15:        if $e$ is not a function return operation then
16:            $p$.abs-dist:= $W(e) + l$.abs-dist
17:        else
18:            $p$ := the corresponding function call location of $l$
19:            $p$.abs-dist:= $FW$($p'$s function name) + $l$.abs-dist
20:        end if
21:        $VS := VS \cup \{p\}$
22:        $Q := Q \cup \{p\}$
23:    end for
24: end while
```

Figure 7.8: Algorithm: `abs-dist`

Bibliography

[1] Jhala, Ranjit, and Rupak Majumdar. "Software model checking." *ACM Computing Surveys (CSUR)* 41.4 (2009): 1-54.

[2] Clarke, Edmund, et al. "Progress on the state explosion problem in model checking." *Informatics.* Springer, Berlin, Heidelberg, 2001.

[3] Clarke, Edmund M., Orna Grumberg, and David E. Long. "Model checking and abstraction." *ACM transactions on Programming Languages and Systems (TOPLAS)* 16.5 (1994): 1512-1542.

[4] Graf, Susanne, and Hassen Saïdi. "Construction of abstract state graphs with PVS." *International Conference on Computer Aided Verification.* Springer, Berlin, Heidelberg, 1997.

[5] Ball, Thomas, Vladimir Levin, and Sriram K. Rajamani. "A decade of software model checking with SLAM." *Communications of the ACM* 54.7 (2011): 68-76.

[6] Ball, Thomas, et al. "Automatic predicate abstraction of C programs." *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation.* 2001.

[7] Ball, Thomas, Andreas Podelski, and Sriram K. Rajamani. "Boolean and Cartesian abstraction for model checking C programs." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Berlin, Heidelberg, 2001.

[8] Ball, Thomas, and Sriram K. Rajamani. "The SLAM project: Debugging system software via static analysis." *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 2002.

[9] Henzinger, Thomas A., et al. "Lazy abstraction." *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 2002.

[10] Henzinger, Thomas A., et al. "Software verification with BLAST." *International SPIN Workshop on Model Checking of Software.* Springer, Berlin, Heidelberg, 2003.

[11] Beyer, Dirk, et al. "The software model checker blast." *International Journal on Software Tools for Technology Transfer* 9.5-6 (2007): 505-525.

[12] Beyer, Dirk, M. Erkan Keremoglu, and Philipp Wendler. "Predicate abstraction with adjustable-block encoding." *Formal Methods in Computer Aided Design.* IEEE, 2010.

[13] Beyer, Dirk, et al. "Software model checking via large-block encoding." *2009 Formal Methods in Computer-Aided Design.* IEEE, 2009.

[14] Beyer, Dirk, and Karlheinz Friedberger. "Domain-independent interprocedural program analysis using block-abstraction memoization." *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 2020.

[15] Beyer, Dirk, and Stefan Löwe. "Explicit-state software model checking based on CEGAR and interpolation." *International Conference on Fundamental Approaches to Software Engineering.* Springer, Berlin, Heidelberg, 2013.

[16] Friedberger, Karlheinz. "CPA-BAM: Block-abstraction memoization with value analysis and predicate analysis." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Berlin, Heidelberg, 2016.

[17] Andrianov, Pavel, et al. "Cpa-bam-slicing: Block-abstraction memoization and slicing with region-based dependency analysis." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Cham, 2018.

[18] Wonisch, Daniel. "Block abstraction memoization for CPAchecker." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Berlin, Heidelberg, 2012.

[19] Clarke, Edmund, et al. "Counterexample-guided abstraction refinement." *International Conference on Computer Aided Verification.* Springer, Berlin, Heidelberg, 2000.

[20] Clarke, Edmund, et al. "Counterexample-guided abstraction refinement for symbolic model checking." *Journal of the ACM (JACM)* 50.5 (2003): 752-794.

[21] Beyer, Dirk, and M. Erkan Keremoglu. "CPAchecker: A tool for configurable software verification." *International Conference on Computer Aided Verification.* Springer, Berlin, Heidelberg, 2011.

[22] Löwe, Stefan, and Philipp Wendler. "CPACHECKER with adjustable predicate analysis." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Berlin, Heidelberg, 2012.

[23] Dräger, Klaus, Bernd Finkbeiner, and Andreas Podelski. "Directed model checking with distance-preserving abstractions." *International Journal on Software Tools for Technology Transfer* 11.1 (2009): 27-37.

[24] Yousefian, Rosa, Vahid Rafe, and Mohsen Rahmani. "A heuristic solution for model checking graph transformation systems." *Applied Soft Computing* 24 (2014): 169-180.

[25] Edelkamp, Stefan, Alberto Lluch-Lafuente, and Stefan Leue. "Trail-directed model checking." *Electronic Notes in Theoretical Computer Science* 55.3 (2001): 343-356.

[26] Li, You, et al. "Steering symbolic execution to less traveled paths." *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages and applications.* 2013.

[27] Ma, Kin-Keung, et al. "Directed symbolic execution." *International Static Analysis Symposium.* Springer, Berlin, Heidelberg, 2011.

[28] Hajdu, Ákos, and Zoltán Micskei. "Efficient strategies for CEGAR-based model checking." *Journal of Automated Reasoning* 64.6 (2020): 1051-1091.

[29] Albarghouthi, Aws, Arie Gurfinkel, and Marsha Chechik. "From under-approximations to over-approximations and back." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Berlin, Heidelberg, 2012.

[30] Bourdoncle, François. "Efficient chaotic iteration strategies with widenings." *Formal Methods in Programming and their Applications.* Springer, Berlin, Heidelberg, 1993.

[31] Hansen, Trevor, Peter Schachte, and Harald Søndergaard. "State joining and splitting for the symbolic execution of binaries." *International Workshop on Runtime Verification.* Springer, Berlin, Heidelberg, 2009.

[32] Beyer, Dirk, Stefan Löwe, and Philipp Wendler. "Sliced path prefixes: An effective method to enable refinement selection." *International Conference on Formal Techniques for Distributed Objects, Components, and Systems.* Springer, Cham, 2015.

[33] Beyer, Dirk, Stefan Löwe, and Philipp Wendler. "Refinement selection." *International SPIN Workshop on Model Checking of Software.* Springer, Cham, 2015.

[34] Albarghouthi, Aws, and Kenneth L. McMillan. "Beautiful interpolants." *International Conference on Computer Aided Verification.* Springer, Berlin, Heidelberg, 2013.

[35] Das, Satyaki, David L. Dill, and Seungjoon Park. "Experience with predicate abstraction." *International Conference on Computer Aided Verification.* Springer, Berlin, Heidelberg, 1999.

[36] Clarke, Edmund, et al. "SATABS: SAT-based predicate abstraction for ANSI-C." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Berlin, Heidelberg, 2005.

[37] Beyer, Dirk, and Andreas Stahlbauer. "BDD-based software verification." *International Journal on Software Tools for Technology Transfer* 16.5 (2014): 507-518.

[38] Jaffar, Joxan, Jorge A. Navas, and Andrew E. Santosa. "Unbounded symbolic execution for program verification." *International Conference on Runtime Verification.* Springer, Berlin, Heidelberg, 2011.

[39] Jaffar, Joxan, et al. "TRACER: A symbolic execution tool for verification." *International Conference on Computer Aided Verification.* Springer, Berlin, Heidelberg, 2012.

[40] Wonisch, Daniel, and Heike Wehrheim. "Predicate analysis with block-abstraction memoization." *International Conference on Formal Engineering Methods.* Springer, Berlin, Heidelberg, 2012.

[41] Craig, William. "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory." *The Journal of Symbolic Logic* 22.3 (1957): 269-285.

[42] McMillan, Kenneth L. "Lazy abstraction with interpolants." *International Conference on Computer Aided Verification.* Springer, Berlin, Heidelberg, 2006.

[43] Wachter, Björn, Daniel Kroening, and Joël Ouaknine. "Verifying multi-threaded software with Impact." *2013 Formal Methods in Computer-Aided Design.* IEEE, 2013.

[44] Albarghouthi, Aws, et al. "Ufo: A framework for abstraction-and interpolation-based software verification." *International Conference on Computer Aided Verification.* Springer, Berlin, Heidelberg, 2012.

[45] Dangl, Matthias, Stefan Löwe, and Philipp Wendler. "CPAchecker with support for recursive programs and floating-point arithmetic." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Berlin, Heidelberg, 2015.

[46] Beyer, Dirk, and Matthias Dangl. "Strategy Selection for Software Verification Based on Boolean Features." *International Symposium on Leveraging Applications of Formal Methods.* Springer, Cham, 2018.

[47] Richter, Cedric, and Heike Wehrheim. "PeSCo: Predicting Sequential Combinations of Verifiers." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Cham, 2019.

[48] Richter, Cedric, and Heike Wehrheim. "Attend and represent: a novel view on algorithm selection for software verification." *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering.* 2020.

[49] Beyer, Dirk, et al. "Conditional model checking: A technique to pass information between verifiers." *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.* 2012.

[50] Beyer, Dirk, et al. "Reducer-based construction of conditional verifiers." *Proceedings of the 40th International Conference on Software Engineering.* 2018.

[51] Tulsian, Varun, et al. "Mux: algorithm selection for software model checkers." *Proceedings of the 11th Working Conference on Mining Software Repositories.* 2014.

[52] Beyer, Dirk, Thomas A. Henzinger, and Grégory Théoduloz. "Program analysis with dynamic precision adjustment." *2008 23rd IEEE/ACM International Conference on Automated Software Engineering.* IEEE, 2008.

[53] Rakamarić, Zvonimir, and Michael Emmi. "SMACK: Decoupling source language details from verifier implementations." *International Conference on Computer Aided Verification.* Springer, Cham, 2014.

[54] Heizmann, Matthias, Jochen Hoenicke, and Andreas Podelski. "Software model checking for people who love automata." *International Conference on Computer Aided Verification.* Springer, Berlin, Heidelberg, 2013.

[55] Dietsch, Daniel, et al. "Ultimate taipan with dynamic block encoding." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Cham, 2018.

[56] Beyer, Dirk, Matthias Dangl, and Philipp Wendler. "Boosting k-induction with continuously-refined invariants." *International Conference on Computer Aided Verification.* Springer, Cham, 2015.

[57] Kroening, Daniel, and Michael Tautschnig. "CBMC–C bounded model checker." International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2014.

[58] Gadelha, Mikhail R., et al. "ESBMC 5.0: an industrial-strength C model checker." *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 2018.

[59] Beyer, Dirk, Matthias Dangl, and Philipp Wendler. "A unifying view on SMT-based software verification." *Journal of automated reasoning* 60.3 (2018): 299-335.

[60] Fenton, Norman, and James Bieman. Software metrics: a rigorous and practical approach. CRC press, 2019.

[61] rpart, https://cran.r-project.org/package=rpart, 2019, Online; accessed 26-Nov-2021.

[62] Mccabe, Thomas. "Cyclomatic complexity and the year 2000." IEEE Software 13.3 (1996): 115-117.

[63] CPAchecker, cpachecker-2.1, gitlab.com/sosy-lab/software/cpachecker/-/tree/tags/cpachecker-2.1 2021, Online; accessed 26-Nov-2021.

[64] *Competition on Software Verification (SV-COMP), Benchmark Verification Tasks.* https://github.com/sosy-lab/sv-benchmarks/tree/svcomp21, Online; accessed 26-Nov-2021.

[65] Cimatti, Alessandro, et al. "The mathsat5 smt solver." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Berlin, Heidelberg, 2013.

[66] Christ, Jürgen, Jochen Hoenicke, and Alexander Nutz. "SMTInterpol: An interpolating SMT solver." *International SPIN Workshop on Model Checking of Software.* Springer, Berlin, Heidelberg, 2012.

[67] Beyer, Dirk. "Software verification: 10th comparative evaluation (SV-COMP 2021)." *Tools and Algorithms for the Construction and Analysis of Systems* 12652 (2021): 401.

[68] Beyer, Dirk, and Alexander K. Petrenko. "Linux driver verification." *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation.* Springer, Berlin, Heidelberg, 2012.

[69] Beyer, Dirk, and Thomas Lemberger. "Software verification: Testing vs. model checking." *Haifa Verification Conference.* Springer, Cham, 2017.

[70] Beyer, Dirk, and Thomas Lemberger. "Symbolic execution with CEGAR." *International Symposium on Leveraging Applications of Formal Methods.* Springer, Cham, 2016.

[71] Burnim, Jacob, and Koushik Sen. "Heuristics for scalable dynamic test generation." *2008 23rd IEEE/ACM International Conference on Automated Software Engineering.* IEEE, 2008.

[72] Beyer, Dirk. "Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016)." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Berlin, Heidelberg, 2016.

[73] Richter, Cedric, et al. "Algorithm selection for software validation based on graph kernels." *Automated Software Engineering* 27.1 (2020): 153-186.

[74] Inverso, Omar, et al. "Bounded model checking of multi-threaded C programs via lazy sequentialization." *International Conference on Computer Aided Verification.* Springer, Cham, 2014.

[75] Kroening, Daniel, et al. "Termination analysis with compositional transition invariants." *International Conference on Computer Aided Verification.* Springer, Berlin, Heidelberg, 2010.

[76] Heizmann, Matthias, Jochen Hoenicke, and Andreas Podelski. "Termination analysis by learning terminating programs." *International Conference on Computer Aided Verification.* Springer, Cham, 2014.

[77] Berdine, Josh, Byron Cook, and Samin Ishtiaq. "SLAyer: Memory safety for systems-level code." *International Conference on Computer Aided Verification.* Springer, Berlin, Heidelberg, 2011.

[78] Kotoun, Michal, et al. "Optimized PredatorHP and the SV-COMP heap and memory safety benchmark." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, Berlin, Heidelberg, 2016.

Acknowledgments in Korean

감사합니다.

Curriculum Vitae in Korean

이 름: 이 낙 원

생 년 월 일: 1986년 09월 04일

학 력

2002. 3. – 2005. 2.	부흥고등학교 (경기 안양시)
2006. 3. – 2013. 2.	건국대학교 컴퓨터공학 (학사)
2013. 3. – 2015. 2.	한국과학기술원 전산학부 (석사)

학 회 활 동

1. 이낙원, 류덕산, 조일훈, 송재근, 백종문, *적합도와 메타학습을 결합한 하이브리드 학습 기반 소프트웨어 신뢰도 예측 모델 선정 기법*, 한국 소프트웨어공학 학술대회, 평창 (한국), 02. 2021.
2. 이낙원, 백종문, *블록 부호화를 통한 지연 술어 추상화 모델 체킹을 빠르게 하기 위한 에러 위치 지향 순회 전략*, 한국 소프트웨어 종합 학술대회, 평창 (한국), 12. 2019.
3. 이낙원, 백종문, *추상 도달가능성 그래프 기반 소프트웨어 모델체킹에서의 탐색전략 고려방법*, 한국 소프트웨어공학 학술대회, 평창 (한국), 02. 2017.

연 구 업 적

1. N. Lee, Y. Kim, M. Kim, D. Ryu and J. Baik, "Directed Model Checking for Fast Abstract Reachability Analysis," in *IEEE Access*, vol. 9, pp. 158738-158750, 2021.
2. 장종인, 이낙원, 류덕산, 백종문, "RESTful 웹 어플리케이션 행위 모델 기반 결함 위치 추정", *정보과학회논문지*, 제 47권 제 11호, pp. 1044–1053, 2020년 11월.
3. 이낙원, 백종문, "추상 도달가능성 그래프 기반 소프트웨어 모델체킹에서의 탐색전략 고려방법", *정보과학회논문지*, 제 44권 제 10호, pp. 1034–1044, 2017년 10월.