Exploring Hybrid Dynamic and Static Techniques for Software Verification

Xueqi Cheng

Dissertation submitted to the faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Michael S. Hsiao, Chair
Dong S. Ha
Patrick R. Schaumont
Yilu Liu
James D. Arthur
Weiguo(Patrick) Fan

Feb. 04, 2010
Blacksburg, Virginia

Keywords: Software Verification, Model Checking, Data Mining, Swarm Intelligence

Exploring Hybrid Dynamic and Static Techniques for Software Verification

Xueqi Cheng

Abstract

With the growing importance of software on which human lives increasingly depend, the correctness requirement of the underlying software becomes especially critical. However, the increasing complexities and sizes of modern software systems pose special challenges on the effectiveness as well as efficiency of software verification. Two major obstacles include the quality of test generation in terms of error detection in software testing and the state space explosion problem in software formal verification (*model checking*).

In this dissertation, we investigate several hybrid techniques that explore dynamic (with program execution), static (without program execution) as well as the synergies of multiple approaches in software verification from the perspectives of testing and model checking. For software testing, a new simulation-based internal variable range coverage metric is proposed with the goal of enhancing the error detection capability of the generated test data when applied as the target metric. For software model checking, we utilize various dynamic analysis methods, such as data mining, swarm intelligence (ant colony optimization), to extract useful high-level information from program execution data. Despite being incomplete, dynamic program execution can still help to uncover important program structure features and variable correlations. The extracted knowledge, such as invariants in different forms, promising control flows, etc., is then used to facilitate code-level program abstraction (*under-approximation/over-approximation*), *and/or state space partition, which in turn improve the performance of property verification.*

In order to validate the effectiveness of the proposed hybrid approaches, a wide range of experiments on academic and real-world programs were designed and conducted, with results compared against the original as well as the relevant verification methods. Experimental results demonstrated the effectiveness of our methods in improving the quality as well as performance of software verification. For software testing, the newly proposed coverage metric constructed based on dynamic program execution data is able to improve the quality of test cases generated in terms of mutation killing — a widely applied measurement for error detection. For software model checking, the proposed hybrid techniques greatly take advantage of the complementary benefits from both dynamic and static approaches: the lightweight dynamic techniques provide flexibility in extracting valuable high-level information that can be used to guide the scope and the direction of static reasoning process. It consequently results in significant performance improvement in software model checking. On the other hand, the static techniques guarantee the completeness of the verification results, compensating the weakness of dynamic methods.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor, Professor Michael S. Hsiao, for his guidance, support and encouragement that helped me throughout my research for this dissertation and prepared me for future challenges. Without his help, this work will never be accomplished.

I would also like to thank all other faculty members serving on my Ph.D committee, Dr. Dong S. Ha, Dr. Patrick R. Schaumont, Dr. Yilu Liu, Dr. James D. Arthur and Dr. Weiguo(Patrick) Fan, for their valuable input and suggestions during my developing the dissertation. I thank Dr. Sandeep K. Shukla and Dr. Layne T. Watson for their helpful comments through my preliminary exam.

I am grateful to all the professors and team members in the Center for Wireless Telecommunications. It's been a really a good experience to participate in such an inspiring project, which also renders me precious practices in relation to my research.

My special appreciation also goes to all the PROACTIVE members, current and alumni, for all the wonderful discussions and enormous helps. With their companionship and friendship, this journey of Ph.D study has been a great pleasure.

I also want to thank the NSF grants 0519959, 0524052 and NIJ grant 2005-IJ-CX-K017 for financial support.

Last, I would like to give my deep thank to my family: parents Yuhuan and Lihui, sister Xuemei, brother-in-law Xin and lovely nephew Stephen, for their love and constant support, without which I would never have come this far. It is to them that I dedicate this work.

Table of Contents

Table of Contents ..... **v**

List of Figures ..... **xi**

List of Tables ..... **xiii**

1. Introduction **1**

1.1 Problem Scope and Motivation 2

1.1.1 Software Testing 2

1.1.2 Software Formal Verification 3

1.2 Contributions of the Dissertation 5

1.2.1 Internal Variable Range Coverage Metric 5

1.2.2 Mining-Based Invariant Extraction for Software BMC 6

1.2.3 Ant-Colony Optimization Guided Under-Approximation 6

1.2.4 Constraint-Reduction-Based Abstraction with Mined Invariants 7

1.2.5 Partitioned CRA 8

1.3 Organization of the Dissertation 8

2. Background and Literature Review **10**

2.1 Software Testing 10

2.1.1 Test Coverage Criteria 10

2.1.2 Automated Test Data Generation 12

2.1.2.1 Search-Based Test Data Generation 12
2.1.3 Domain-Based Partition Testing 13
2.1.3.1 Boundary-Value Analysis 14
2.1.4 Mutation Testing 16
2.1.4.1 Mutation Operators 17
2.2 Model Checking 17
2.2.1 Transition Systems and Temporal Logic Properties 18
2.2.2 Model Checking Algorithms 19
2.2.2.1 Symbolic Model Checking 20
2.2.2.2 Bounded Model Checking 20
2.3 Software Model Checking 21
2.3.1 Abstraction Techniques 22
2.3.1.1 Constraint-Reduction-Based Abstraction (CRA) 24
2.3.1.2 Predicate Abstraction 25
2.3.2 Counter-Example Guided Abstraction Refinement (CEGAR) 27
2.3.3 SAT-Based Software Bounded Model Checking 28
2.4 Invariant Discovery Techniques 30
2.4.1 Static Program Analysis 30
2.4.2 Data Mining 33
2.4.2.1 Basic Definition of Association Rule 33
2.4.2.2 The A-Priori Algorithm 34
2.4.3 Invariant Extraction and Model Checking 36
2.5 Swarm Intelligence 37

2.5.1 Ant Colony Optimization....38

3. Internal Variable Range Coverage Metric and Test Generation Model....40

3.1 A Motivating Example.... 41

3.2 Internal Variable Partition Testing.... 42

3.2.1 Critical Internal Variable Selection....43

3.2.2 Dynamic-Program-Execution-Based Variable Range Definition....43

3.2.3 Range Partition and Range Merge....44

3.2.4 Internal Variable Range Coverage Metric....45

3.3 Automated Test Generation Model.... 46

3.3.1 Sub-Range Transition System (SRTS)....47

3.3.2 Goal-Oriented Search Algorithm....49

3.4 Experimental Results.... 50

3.4.1 Experiment Setup....50

3.4.2 Sub-Range Combination Coverage....51

3.4.3 Mutant Coverage....52

3.4.4 Fault Detection Capability....53

3.5 Conclusions.... 54

4. Mining-Based Invariant Extraction for Bounded Model Checking....55

4.1 Motivation.... 55

4.2 Mining-Based Software Model Checking Framework.... 57

4.3 Property Invariant Mining and Rule Reduction Algorithms.... 61

4.3.1 Property Invariant Mining....61

4.3.2 Mining Rule Reduction Algorithms....66

4.3.3 Ranking Scheme 68
4.4 Experimental Results 69
4.4.1 Experiment Setup 69
4.4.2 Data Mining and Rule Reduction 69
4.4.3 Model Checking 71
4.5 Conclusions 73

5. Ant Colony Optimization Directed Under-Approximation Based Program Abstraction 74

5.1 Motivation 75
5.2 Related Work 77
5.2.2 Program Abstraction Based Verification 77
5.2.3 Dynamic Execution Guided Software Model Checking 78
5.3 ACO-Directed State Space Exploration 78
5.3.1 ACO-Directed State Space Exploration Model 79
5.3.2 Stochastic Promising Structure Construction 81
5.3.3 Pheromone Update 82
5.3.4 Desirability Update 84
5.3.5 Daemon Actions 85
5.3.5.1 Local Optimum Detection 85
5.3.5.2 Additional Pheromone Adjustment 86
5.4 Bounded Model Checking through Iterative Under-Approximation Based Program Abstraction 87
5.5 Experimental Results 90

5.6 Conclusions 93

6. Constraint-Reduction-Based Abstraction with Single Mined Invariants.....94

6.1 Introduction 94

6.2 Constraint-Reduction-Based Abstraction with Single Mined Invariants 96

6.2.1 Potential Single Invariant Mining 96

6.2.2 Constraint-Reduction-Based Abstraction over Single Invariants 96

6.2.3 Potential Single Invariant Selection 98

6.3 CEGAR with Constraint-Reduction-Based Abstraction 99

6.3.1 CEGAR-CRA Framework 99

6.3.2 Counter-Example Guided Refinement.....100

6.4 Experimental Results 103

6.5 Conclusions 106

7. Local Invariant-Directed Path-Level Partitioning with CRA 107

7.1 Introduction 107

7.2 CPI-Directed Path-Level Partitioning with CRA..... 110

7.2.1 Potential Complex Property Implication Mining.....110

7.2.1.1 Branch Coverage Directed Dynamic Program Execution 111

7.2.2 Path-Level Partitioning with CRA.....116

7.2.3 Invariant Selection and Partitioning Strategy 119

7.3 CEGAR with Partitioned CRA 121

7.4 Experimental Results 122

7.5 Conclusions 125

8. Conclusions and Future Work 126

8.1 Conclusions 126

8.2 Suggestions for Future Work 129

8.2.1 Partitioning with Non-Uniform Abstraction Techniques 129

8.2.2 Program Structure Directed Invariant Mining for Abstraction 130

8.2.3 Reusing the Reasoning Results 131

Bibliography 133

List of Figures

Figure 2.1 An Example of PCT Coverage 11

Figure 2.2 An Example for Input Domain Partitioning 14

Figure 2.3 Test Points for a Two-Dimensional Linear Border 15

Figure 2.4 An Example of 1×1 Domain Testing Strategy 16

Figure 2.5 Examples for Under-Approximation Based Software Abstraction 23

Figure 2.6 Constraint-Reduction-Based Abstraction (CRA) Example 25

Figure 2.7 Predicate Abstraction Example 26

Figure 2.8 CEGAR Paradigm 28

Figure 2.9 Software Bounded Model Checking (CBMC) 29

Figure 2.10 An Example for Live Variable Analysis 31

Figure 2.11 Fix-Point Computation 32

Figure 2.12 General ACO Algorithm 38

Figure 3.1 A Motivating Code Example for Range Partitioning 41

Figure 3.2 Framework of Internal Variable Partition Testing 42

Figure 3.3 Sub-Range Transition System SRTs for Internal Variables V_i and V_j 48

Figure 3.4 Benchmark Program Structures 51

Figure 4.1 Example Code Segment 1 (Loop) 56

Figure 4.2 Example Code Segment 2 (Control) 56

Figure 4.3 Mining-Based Software Model Checking Framework..... 58

Figure 5.1 Example for Property Falsification 76

Figure 5.2 ACO-Directed State Space Exploration Model..... 80

Figure 5.3 BMC Algorithm through Iterative UAPA..... 87

Figure 6.1 Constraint-Reduction-Based Abstraction over P..... 97

Figure 6.2 CEGAR-CRA Framework..... 99

Figure 6.3 CEX-Guided Refinement Algorithm..... 101

Figure 6.4 A CEX-Guided Refinement Example for CRA 102

Figure 7.1 A Motivating Example of Partitioned CRA 108

Figure 7.2 Input Model for Branch Coverage Directed Input Generation..... 112

Figure 7.3 Local-Search-Based Input Generation Algorithm..... 113

Figure 7.4 Probabilistic Model Between Inputs and Branch Distance 115

Figure 7.5 An Example of Path-Level Partitioning Directed by PCPI 116

Figure 7.6 An Example for Path-Level Partitioning with CRA..... 118

Figure 7.7 Partitioning Strategy for the Selected CPI Set 119

Figure 7.8 CEGAR with Partitioned CRA..... 121

List of Tables

Table 2.1 An Example Data Set for Association Rule Mining 33

Table 2.2 Frequent 1-Itemset (s_{th}=0.5) 34

Table 2.3 Frequent 2-Itemset (s_{th}=0.5) 34

Table 2.4 Association Rule Derivation (c_{th}=0.8) 35

Table 3.1 IVR Coverage Results 52

Table 3.2 Mutation Operators 52

Table 3.3 Mutants Killing Results 53

Table 4.1 An Example Database for Property Invariant Mining 62

Table 4.2 Frequent 1-ItemSet Computation for SLP Mining (s_{th}=0.5) 63

Table 4.3 Association Rule Derivation for SLP Mining 63

Table 4.4 Frequent 2-itemset Computation for CPI Mining (s_{th}=0.5) 65

Table 4.5 Association Rule Derivation for CPI Mining 65

Table 4.6 Results for Mining and Rule Reduction Algorithms 70

Table 4.7 Property Rankings for BubbleSort 71

Table 4.8 Model Checking Results with Added Invariants 72

Table 5.1 Bounded Model Checking Results with UAPA 90

Table 5.2 Random-directed vs. ACO-directed Abstraction 92

Table 6.1 Bounded Model Checking Results with Single Invariant-Directed CRA 104

Table 7.1 Branch Distance Fitness Computation..... 114

Table 7.2 Bounded Model Checking Results with Partitioned CRA..... 123

Chapter 1

Introduction

Nowadays, software systems are playing an increasingly important role in our lives both economically and socially. On one hand, software correctness is essential; poor quality or failures are no longer acceptable and can even result in catastrophic losses. On the other hand, where there is software, there are bugs. The fallibility of its human designers and the inherently complex nature of software make software systems prone to faults. In order to address this dilemma, it is necessary but also challenging to accompany the software development process with effective and efficient software reliability assurance activities with the goal of early detection of both design and program errors. *Software verification* is such a broad discipline with the goal of assuring that the software under verification fully satisfies all the expected requirements. The scope of software verification could span over various aspects of the software, including functionality, performance, usability, security, etc. In this dissertation, by verification, we especially refer to verifying functionality requirements specified in the software design specification.

The current practice in the software design industry is to use *testing* to verify software, which is a *dynamic* process of executing a program with the intent of finding errors. However, such testing is typically not exhaustive and therefore is not sufficient to guarantee correctness − it cannot show the absence of errors. Such a problem can to a great extent be mitigated by *formal verification* of software, which rigorously and *statically* proves the correctness of computer programs against the design specification through an implicit, yet complete search over the entire state space without executing the program. Formal methods guarantee the compliance of the code to the specification but at the cost of complexity and performance, especially for large and intricate software programs.

In this dissertation, we explore the utilization of dynamic program execution in software verification mechanisms, including both dynamic (testing) and static (formal verification) methods. At the same time, we would also discuss the applicability of hybrid solutions that combine dynamic program execution, testing and verification techniques together to further enhance the effectiveness and performance of software verification.

1.1 Problem Scope and Motivation

1.1.1 Software Testing

Software testing is an activity of identifying the quality of programs. In particular, we focus on functional testing, which establishes the correct operations of the software against expected behaviors defined in the design specification. Although testing has been widely deployed during the process of software development, the increasing complexities and uncertainties of the software systems have made it an expensive, labor-intensive and error-prone activity. Studies have shown that testing accounts for about half the total cost of software development and maintenance [53], which places significant demands on *automated* and *effective* software testing strategies. The major process of the functional testing involves exercising the program with designed test cases to validate whether the software behaves in accordance to the requirements until certain coverage threshold is reached in terms of code coverage. Therefore, test case generation and code coverage

metric are two critical factors for achieving automation and effectiveness in software testing.

Partition testing [7] represents a fundamental class of sub-domain-based functional testing strategies. The main idea is to divide the overall input domain D into a collection of k sub-domains D_1, D_2 ,…, D_k, either disjoint or overlapping, based on specification and/or program structure. A sub-domain usually consists of inputs that satisfy certain equivalent properties or relations depending on the specific testing coverage applied, e.g., statement coverage, branch coverage, path coverage, etc. The partition testing then selects one or more test cases from each sub-domain and tests them accordingly.

It is apparent that the effectiveness of the partition testing highly depends on how the input sub-domains are defined. An ideal domain partition would be to produce homogeneous sub-domains, that is, all the test points from a sub-domain cause the program to succeed or all cause it to fail [54]. However, in practical situations, the simple partition strategies for sub-domain coverage are found to be inadequate, particularly for programs specifying or implementing the sub-domains involving numeric input variables [24], sub-domain definition based merely on program structures may be too coarse to detect data-related faults. Moreover, for programs with complex work flows, where the direct relationship between external input variables and the internal implementation and logic may not be established easily, it may consequently incur higher input domain partition overhead. In other words, domain partition design and coverage metric turn out to be the most challenging tasks for the test case generation of partition testing.

1.1.2 Software Formal Verification

Formal verification of software is a process of using formal methods to prove the correctness of a computer program with respect to its design specification. Model checking, as one of the formal methods, explores all the paths in the model and has been very successful for verifying finite-state systems. When applicable, it automatically provides complete proofs of correctness, or explains, via counter-examples, why a system is not correct. Despite the great success the model checking gained in the verification of

hardware designs, verifying software systems, however, is a different and harder problem because such systems are inherently infinite-state: many variables are (essentially) infinite-domain and the heap is of unbounded size. Cost and scalability remain the major obstacles of software model checking due to the extremely large state space.

Bounded Model Checking (BMC) [1, 2] is an automatic verification technique that checks the temporal logic property based on unfolding the system's transition relation for a given finite depth. BMC first translates the design under verification that violates certain specification(s) into a propositional formula and thus converts the problem into a propositional satisfiability (SAT) problem. Then, a SAT-solver is used to search for a counterexample whose length is bounded by a pre-defined bound *k*. By leveraging the advances in modern SAT-solvers, SAT-based BMC has become a scalable verification alternative to the Binary Decision Diagram(BDD)-based symbolic model checking and gained wide acceptance for the automatic analysis of finite state systems. In addition to the success in hardware community, recent research efforts [6, 37] have also applied it as an effective verification method on inherently infinite state software systems via transforming the program code into a Boolean representation with bounded unwinding length. However, BMC essentially reduces the size and complexity of the design under verification in the form of limited unwinding depth, which may not be a thorough solution for designs with industrial-size or the unwound parts already involve complex structures.

Hence, effective state reduction techniques are still needed to alleviate the potential state space explosion problem of software formal verification. Abstraction techniques, which reduce system complexity by removing irrelevant information while retaining only useful portions, have provided promising direction towards countering the problem. Program abstraction refers to simplifying the original code with either reduced control/data structure or coarser details. As a result, the abstracted program with decreased complexity tends to show higher scalability and performance than the original in terms of property checking. Nevertheless, taking the complexity of software into account, to what extent can the strengths of program abstraction be taken advantage of still relies heavily on *how* and *where* the abstraction is applied within the code.

In recent years, hybrid approaches that synergize multiple analysis techniques with formal verification have drawn growing interests [64, 65, 66, 67]. Among these approaches, the dynamic techniques associated with program execution usually exhibit complementary benefits compared to the static techniques, such as efficiency, flexibility, straightforwardness, etc. Thus, the hybrid techniques demonstrate promising capability of exploring useful and desired knowledge on the sophisticated program structure for software model checking, while keeping the overhead feasible.

1.2 Contributions of the Dissertation

In this dissertation, our contributions cover two aspects of software verification: 1) For software testing (Section 1.2.1), we aim to raise the quality of testing in error detection through an enhanced coverage metric. Different from the existing code coverage, like statement coverage, path coverage, etc., which mainly associates with the static control flow structures, our metric explores the alternative *data* aspects of variables according to the dynamic program execution profiles, such that value-related computation errors can also be captured; 2) For software formal verification (Section 1.2.2−1.2.5), we propose hybrid techniques combining various dynamic approaches with static formal verification to leverage the complementary power from both. On one hand, the results of dynamic analysis can be utilized to facilitate the program abstraction as well as to speedup the property checking, which might otherwise be expensive for pure static proving; on the other hand, the formal methods guarantee the completeness of verification results, which usually can't be expected from the dynamic program execution.

1.2.1 Internal Variable Range Coverage Metric

We present a new sub-domain-based testing strategy which is associated with a new coverage metric called Internal Variable Range (IVR) coverage [94]. Unlike the traditional input domain partition techniques, we apply partitioning on the value ranges of critical internal variables in a program. A dynamic-program-execution-based approach is proposed for range analysis and range definition according to the variable data value distribution as well as the program structure information related to the internal variables. In addition, we exploit a sub-range transition system to facilitate the internal variable

range coverage oriented automated test data generation process. Our experimental results showed the effectiveness of our model in test set generation targeting the internal variable range coverage compared to random approaches. In addition, such a test set also exhibits better quality in terms of mutation killing, especially for applications that are more arithmetic computation related.

1.2.2 Mining-Based Invariant Extraction for Software BMC

Studies showed that invariants play critical roles in the process of automated formal verification; however, SAT-based BMC has the limitation that requires translating high-level program into one large propositional formula, which could place extra burden to the reasoning process by re-discovering crucial high-level constraints at the Boolean level. Motivated by such a gap, we propose a novel framework for software formal verification that uses data mining to help improve the performance of software bounded model checking [95]. Data mining is the process of analyzing and extracting useful but implicit knowledge from large amounts of data. In our mining-based framework, we use an association rule mining [10, 11] approach to discover potential high-level property invariants according to the dynamic execution data of the software programs. Optimization algorithms are then applied to minimize the number of properties by removing both redundant and false positives being discovered. As a result, the cost of the potential property verification is reduced as well. Finally, the verified mined properties are added as constraints to the original program to help reduce the search space of the model checking. The experimental results demonstrated that our approach could lead to up to an order of magnitude of improvement on the performance of software bounded model checking.

1.2.3 Ant-Colony Optimization Guided Under-Approximation

Ant Colony Optimization (ACO) is a class of meta-heuristic, bio-inspired swarm intelligence algorithms that simulate the foraging behaviors of real-world ants. By leveraging of power of ACO in finding approximate solutions for optimization and search problems, we propose an ACO-directed under-approximation based abstraction technique to facilitate the construction of path-level code abstraction for software model checking

[96]. In this work, a set of artificial ant agents are sent out to explore the concrete state space of the program as well as the corresponding program control flow with the aim of negating the target property to be checked. By means of leaving pheromone trails along the software paths traveled, ant agents are able to cooperate with each other indirectly towards achieving the common goal. In addition, centralized daemon actions are taken periodically to pull the ant solutions out of tendencies of "local optimum". Therefore, eventually, the traces with biased pheromone levels tend to reveal the most promising program structures in terms of refuting the target property, which can further be used to construct program abstraction via neglecting less useful code portions. Subsequently, an abstraction-based software model checking framework is also proposed to guide the verification process over the reduced program through iterative refinement, until it contains enough details to find a real counterexample or verify the property. Experimental results on various property falsification instances demonstrate the strength of ACO in guiding the construction of an effective under-approximated program abstraction, as well as in the performance savings of software bounded model checking.

1.2.4 Constraint-Reduction-Based Abstraction with Mined Invariants

By further exploring the feature of mined invariants in depicting essential correlations among program variables from the work described in Section 1.2.2, we propose a constraint-reduction-based program abstraction (CRA) for software model checking under the guidance of learned invariants. Rather than simply applying the invariants as additional constraints, we use them to build an over-approximation of the program behaviors by relaxing the original relations which are responsible for the inference of the invariants. In other words, we free related variables involved in the invariants, while at the same time, appending the invariants to where they belong to avoid over-freeing the state space. In order to maximally expand the power of invariant-directed CRA, heuristics are also discussed on how to select effective invariants to be applied. We present a new counterexample guided abstraction refinement (CEGAR) framework to gradually refine the coarse abstraction by adding back the removed relations according to the analysis of the spurious counterexamples until reaching a verification conclusion. Significant performance improvement can be achieved using the invariant-directed CRA abstraction

according to the experiments on both academic and real-world programs. It also outperforms the work from Section 1.2.2 where invariants are only added as additional constraints with no over-approximation applied.

1.2.5 Partitioned CRA

Considering that this invariant-directed CRA requires the underlying invariants to hold globally over the entire program, we further introduce the notion of partitioning to the abstraction to break this limitation. A path-level partitioning scheme is proposed according to the mining results on invariants inferred from certain local flow structure, rather than global linear relations. It divides the state space of program into functionally disjoint subspaces in the form of control paths, and separate CRA abstraction can then be constructed and refined within each partition independently. The path-level partitioned CRA expands the applicability and effectiveness of global invariant-directed CRA, as it enables more local invariants to be involved in the abstraction applying over a set of more aggressively simplified programs (i.e., partitions) such that the complexity of the program under verification can be further reduced. At the same time, it potentially offers an abstraction paradigm with higher flexibility by allowing the construction of non-uniform abstraction methods over different partitions. In combination with the CEGAR framework, the partitioned abstraction technique is able to greatly enhance the performance of software bounded model checking due to the drastically reduced program complexity brought by the integration of partitioning and abstraction. Moreover, it also demonstrates higher capability of performance savings than the global invariant-directed CRA abstraction in most of the experiments.

1.3 Organization of the Dissertation

The remainder of the dissertation is organized as follows. In Chapter 2, we provide background knowledge and a review of literature in the area of software testing, software model checking as well as invariant discovery and swarm intelligence techniques. In the next five chapters (3–7), we present various hybrid techniques with the purpose of improving the quality as well as performance of software verification. Our dynamic-program-execution-based internal variable range coverage metric and test generation

model are described in Chapter 3. In Chapter 4, we propose a new mining based invariant extraction technique to help reduce the state space of software model checking. A novel Ant Colony Optimization (ACO) directed under-approximation based abstraction technique is given in Chapter 5 to aggressively speedup the property falsification, and Chapter 6 presents a new CEGAR framework built on invariant-directed constraint-reduction-based abstraction (CRA) technique. Chapter 7 further extends the work in Chapter 6 and applies the CRA over partitions of the program control flows. Finally, in Chapter 8, we summarize the dissertation by concluding our discoveries and discussing the future work.

Chapter 2

Background and Literature Review

In this chapter, we introduce the background knowledge of software verification techniques: software testing and model checking, together with invariant discovery approaches and swarm intelligence. A review of literature in each area will also be provided.

2.1 Software Testing

Software testing is oriented toward the '*detection*' of faults in a program in a minimum amount of time and with a minimum amount of effort. It usually intentionally attempts to make the software go wrong to determine if the program behaviors happen when they shouldn't or don't happen when they should. A secondary benefit of testing is that it demonstrates that the software appears to be working as stated in the specifications. The data collected through testing can also provide an indication of the software's reliability and quality.

2.1.1 Test Coverage Criteria

Test coverage (or code coverage) is a measure of how thoroughly the tests exercise the code base, which can be considered as an indirect measure of test quality because it evaluates the degree to what the tests cover the target code. Therefore, code coverage can be used to guide the testing process and help devising test cases/inputs or configuration

sets towards the direction of covering those untested parts of the program. There is a number of different ways of measuring code coverage. The fundamental ones include: 1) *statement coverage* (i.e., has each line of the source code been executed?), 2) *decision/branch coverage* (i.e., has each evaluation point (such as a true/false decision) been executed?) and 3) *path coverage* (i.e., has every possible route through a given part of the code been executed?).

New coverage metrics were also proposed to complement/enhance the existing criteria. Modified Condition/Decision Coverage (MC/DC) [46] is a widely applied variation of decision coverage. It enhances the decision/branch coverage criterion by requiring that each Boolean sub-expression to independently affect the outcome of the decision, such that the effect of each condition can be tested relative to the other conditions.

Ball [47] proposed a new form of coverage metric called Predicate-Complete Test (PCT) coverage with the goal of covering every reachable observable state in a program, that is, an evaluation of the *n* predicates under some state at a program statement. Considering the program fragment given in Figure 2.1[47], the two if-statements correspond to a predicate set containing four atomic predicates: {p_1=(x<0), p_2=(y<0), p_3=(w<x), p_4=(z>y)}. PCT requires that every Boolean combination over the set be tested at every statement in the program (6 in this case: L_1 thru L_6). For example, {L_1, TTFT} and {L_2, TFFT} represent two such observable states.

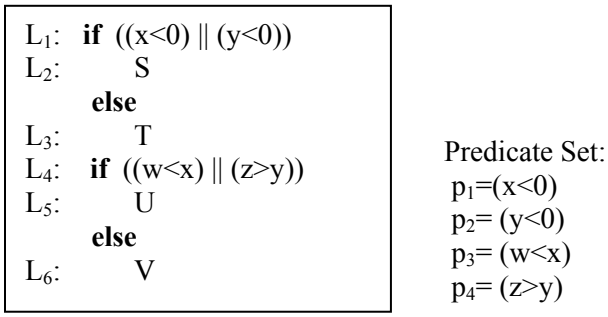


Figure 2.1 An Example of PCT Coverage

The PCT coverage subsumes many existing control-flow coverage criteria such as statement, branch and multiple condition coverage. Taking Figure 2.1 as an example, complete PCT coverage clearly implies full coverage of statements and branches. While

the multiple condition coverage only requires that every Boolean combination over {p_1, p_2} be tested at L_1 and that every Boolean combination over {p_3, p_4} be tested at L_4, which can be subsumed by PCT coverage as well. In addition, PCT can be viewed as an alternative to path coverage that has its "exhaustive" quality but induces a finite (rather than infinite) state space.

Other than the classical code-based coverage criteria, investigations have been made on other types of coverage criteria to evaluate the test adequacy from different perspectives or for different applications. For example, Memon et al. [48] presented new coverage criteria in terms of GUI events and their interactions to help determine whether a GUI has been adequately tested. A new family of model-based coverage criteria was proposed in [49], based on formalizing boundary-value testing heuristics extracted from the specification. The boundary coverage criteria form a hierarchy of data-oriented coverage criteria, and can be applied to any formal notation that uses variables and values.

2.1.2 Automated Test Data Generation

Test data generation in program testing is the process of identifying a set of program inputs which satisfies selected testing coverage criteria. The cost of software testing can be greatly reduced if the process of test data generation is automated. Due to the complexity, diversity and uncertainty of the software systems, automated generation for high-quality and yet maintainable test suite remains an area of great research interest.

2.1.2.1 Search-Based Test Data Generation

Meta-heuristic search is a dynamic technique that has been widely applied on automatic test data generation. It generally involves the transformation of test criteria to objective functions, which can be used to direct the test data search into potentially promising areas of the search space. In practice, a fitness value is associated to the objective function for each input of the system and the test data are incrementally modified to optimize their fitness values. In this context, the test generation problem becomes an optimization problem.

A representative meta-heuristic search algorithm is *local search based* [38, 39, 40, 101]. Using an initial solution randomly chosen from the search space as a starting point, local search looks for a better solution by investigating its neighborhood with respect to the objective function. If a better solution is found, then the current solution is replaced with the better one, this process is repeated until no improved solution can be found. The local search based approach is simple and fast; however, it may be trapped in locally optimal points.

Some of the problems in local search such as sensitive to small change in input values can be solved using simulated annealing based techniques. For example, Diaz et al. in [41] applied a similar Tabu search to improve the search efficiency by looking at the next k neighbors of the input vector. Then, selected the best solutions among the neighbors, while maintaining tabu lists of visited neighbors as well as frequently visited neighbors that are forbidden, until the target branch has been reached.

Genetic and evolutionary algorithms form another major category of search based test data generation [42, 43, 44, 45]. These algorithms are inspired by evolutionary ideas of natural selection and reproduction. The evolution starts from a population of completely random individuals and happens in generations. A fitness function evaluates each solution to decide whether it will contribute to the next generation of solutions. Then, through operations analogous in evolutionary biology, such as inheritance, mutation, crossover, etc., the algorithm creates a new population of candidate solutions to be used in the next generation. Different techniques that apply genetic algorithms to test data generation usually differ in the way of fitness function construction.

2.1.3 Domain-Based Partition Testing

Domain-based partition testing provides an effective way of test generation. Its essence is in the *partitioning* of the input *domain* into a finite number of sub-domains, and considering the input values in each domain as an equivalence class. Figure 2.2 shows an example of white-box based input domain partitioning. Integer x is the input variable of the program snippet shown on the left. According to the internal structure of the program, in particular, the conditions of the conditional statements, we can partition the

input domain of x into the following four sub-domains: D_1: $(-\infty, 1)$, D_2: $[1, 6]$, D_3: $(6, 12]$, D_4: $(12, \infty)$, such that all the valid input values within each sub-domain form an equivalence class in the form of taking the same path.

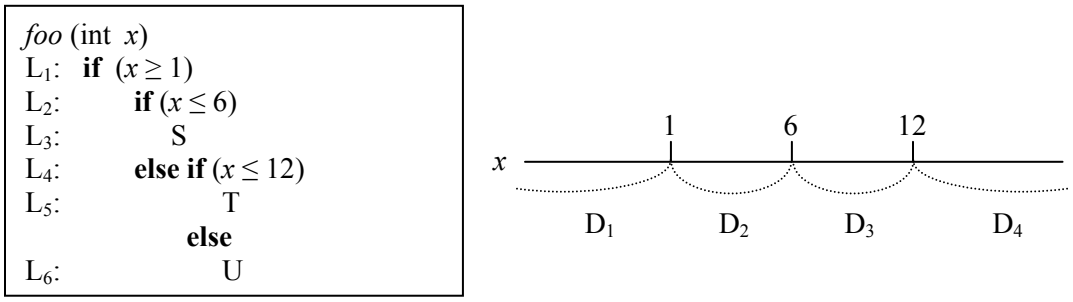


Figure 2.2 An Example for Input Domain Partitioning

Domains can be exhaustively tested and covered by selecting a minimal set of representative value(s) in each domain. Therefore, the key of partition testing is how to achieve an ideal partitioning of the input domain conveniently as well as effectively. Cai et al. introduced the idea of dynamic partitioning in [50]. The test suite or the input domain of the software under test is partitioned into several sub-domains whose members are dynamically adjusted during testing by using feedback information such as the testing data collected on-line.

2.1.3.1 Boundary-Value Analysis

Experience shows that input values at the extreme ends of, and just outside of, input domains tend to exercise corner-case errors in the system functionality. Boundary value analysis, as a complementary technique of the equivalence partitioning, requires one or more boundary values selected at each sub-domain partition as representative test cases.

In order to detect border-shift errors, White and Cohen [105] proposed a so called N×1 strategy to select test points at the boundary of a selected path domain. Two types of test points were defined: 1) an ON test point is a value that lies on the given boundary; 2) an OFF point is a value that lies slightly off the boundary and on the open side of it. For a closed inequality border (i.e., defined by a relational operator containing inequalities, ≥ or ≤) in a two-dimensional domain space, the N×1 strategy requires two ON and one OFF

points to be selected, as shown in Figure 2.3(a) [51, 105]. The ON points are used to identify the given border, and the OFF point is used to assure that the correct border (i.e., the dotted line in Figure 2.3(a)) does not lie on the open side of the given border. This strategy can be easily extended to N-dimensional cases, which requires N ON points to define the N-1 dimensional border.

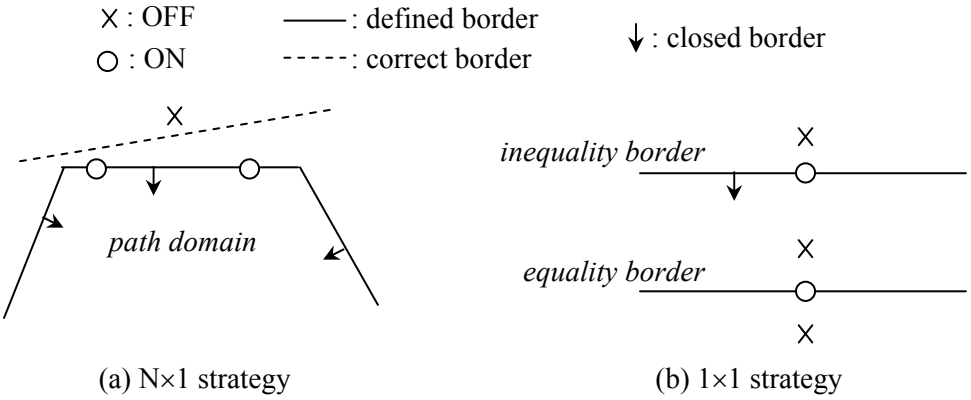


Figure 2.3 Test Points for a Two-Dimensional Linear Border

Jeng et al [51] further proposed a simplified 1×1 domain testing strategy and reported that an inequality border(defined by ≥, >, ≤ or <) can be adequately tested by using only two points of test input domain, one ON point and one OFF point. As shown in Figure 2.3(b), the ON point can be anywhere on the given border. It does not even have to lie exactly on the given border. All that is necessary is that it satisfies the path condition associated with the border. The only requirement for the OFF point is that it be as close to the ON point as possible, but it should lie outside the border. For each strict equality condition (=), one ON and two OFF points are chosen, one slightly less than the conditional value and one slightly greater than the value.

Figure 2.4 gives an example of test case selection using the 1×1 strategy. The inequality boundaries (bold lines) defined by the two if-statements can each be covered by two test cases, containing one ON and one OFF points, respectively. Note that whether the OFF point is *inside* or *outside* the path domain is determined by the closeness of the boundary and is opposite to that of the corresponding ON point. For example, the ON point T_1 for the closed boundary case in Figure 2.4 is inside the domain, while the OFF point T_2 is outside, the other open boundary case is the opposite.

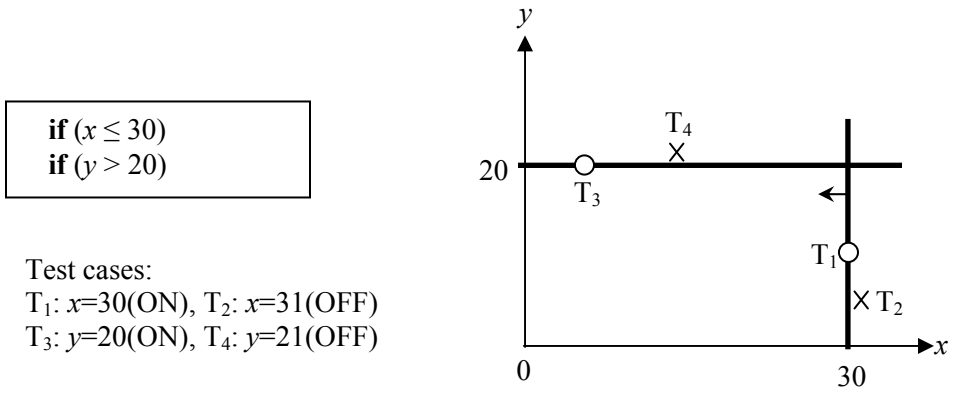


Figure 2.4 An Example of 1×1 Domain Testing Strategy

2.1.4 Mutation Testing

Mutation testing is a fault-based testing technique that is based on the assumption that a program is well tested if all simple faults are predicted and removed. Complex faults are coupled to simple faults in such a way that a test that detects all simple faults would detect most complex faults [52]. Therefore, mutation testing is often used to check the adequacy and evaluate the quality of test suite generated.

This is done by first seeding the original program with a small change, called a mutation, which introduces a fault. Each faulty program is called a *mutant*, with one fault injected. A mutant represents a likely fault, a mistake that programmer could have made, or a typical testing heuristic which is unsatisfied (such as executing each branch). If a mutant is functionally equivalent to the original program, it is called an *equivalent mutant*. After creating a set of mutants, for each non-equivalent mutant, mutation testing would compare the outputs of the mutant with the output of the original program for each test data to check if the test data is able to catch the errors. If the test results are different it shows that this mutation can be detected if made by error, we say the mutant is *killed*. If the test results are not different, it suggests that more test cases might be needed. The percentage of non-equivalent mutants killed is the test suite's *mutation coverage*. Mutation testing can increase the thoroughness of the testing, and hence the quality of the test suite.

2.1.4.1 Mutation Operators

A mutation operator is a rule to be applied to a program to create mutants, so that the slightest change in which may cause the program to function incorrectly. Here are some examples of possible mutation operators:

a) *Replace each operand by every other syntactically legal operand:* for example, a mutation of statement '*A=B*' could be '*A=C*', where *A*, *B*, *C* are all scalar variables in the program.

b) *Modify expressions by replacing operators and inserting new operators*: for example, '*if (x < y)*' can be mutated into '*if (x ≤ y)*'.

c) *Delete entire statement*, etc.

The effectiveness of mutation testing depends heavily on the types of faults that the mutation operators are designed to represent.

2.2 Model Checking

In the last two decades, model checking, pioneered by Clarke et al. [20] and independently, Quellie et al. [22], has emerged as a promising and powerful approach to automatic verification of systems [97]. Typically, model checking is a process of systematic exploration of the possible behaviors of a finite state concurrent system [21] to determine whether the system (abstractly represented as a model) satisfies a specified property (e.g., absence of deadlocks, liveness, invariants, etc., expressed as a logical or temporal formula). If the property is not satisfied, the model checker provides a counterexample − an execution path in the state space of the system that violates the property, which is also valuable for debugging. Model checking has two major advantages over other techniques: firstly, the model checking process is completely automatic, and secondly, the counterexample usually uncovers subtle errors that would be difficult to discover by others.

2.2.1 Transition Systems and Temporal Logic Properties

In order to verify the correctness of a system, the first step is to construct a formal model for the system. The model should capture those properties that must be considered to establish correctness, and at the same time, should abstract away those details that do not affect the correctness of the checked properties but make verification more complicated [21]. A common framework for the representation of reactive systems is provided by the concept of *transition systems* (or *Kripke structures* if the transition systems can be further simplified according to the needed labels sufficient to reason about the system). A transition system uses a state transition graph to capture the behaviors of the system, where the nodes represent system states and the edges represent the possible transitions between states.

Definition 2.1 Kripke Structure. A Kripke structure over a set of atomic proposition AP is a 4-tuple: $T = (S, S_0, R, I)$, where:

- S is a finite set of *states*,
- $S_0 \subseteq S$ is a set of *initial states*,
- $R \subseteq S \times S$ is a *transition relation.*
- I: $S \rightarrow 2^{AP}$ is an *interpretation* that labels each state with the set of atomic propositions true in that state

The sequential behavior of a transition system can be defined by *paths* (*or executions*), which is a finite or infinite sequence of $s_0 \longrightarrow s_1 \longrightarrow s_2 \cdots\cdots \longrightarrow s_n$, such that $s_0 \in S_0$ and $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. Basically, the model checking algorithms examine the conformance of the transition system with a specified property behavior over paths.

Properties specify the temporal behaviors that we are interested to check in a transition system. It asks questions such as "Will the system be stuck at a deadlock state?", "Is this path in the program eventually been executed?", etc.

Temporal logic [23] is a formalism for describing sequences of transitions between states

in a reactive system [21], which makes it a convenient language to formally express these complex properties. The logic uses atomic propositions, Boolean connectives and temporal operators to build up complicated expressions describing properties of states ordering in time. Two common variants in temporal logics are: *Linear Temporal Logic* (LTL) and *Computation Tree Logic* (CTL) [25], which represents two different views of time model: *linear-time* and *branching-time*, respectively. For further information on the two temporal logics, we refer the readers to [21, 23, 25].

Generally, there are two common categories of properties: safety and liveness properties. Informally, a safety property asserts that "*something bad never happens during execution of a system*". Invariants, deadlocks, reachability, etc. are some general examples of safety properties. On the other hand, a liveness property claims that "*something good will eventually happen*". In this dissertation, we'll focus on safety property verification.

2.2.2 Model Checking Algorithms

Given a transition system *M*, in the form of variations on finite state automata, and a property φ, in the form of variations on temporal logic, the model-checking problem decides whether the model *M* semantically satisfies the specification:

$$M \models \varphi \text{ (} M \text{ is a model of } \varphi\text{)}$$

Originally, the model checking algorithm was implemented by means of "explicit-state" techniques, where individual states of the transition system are analyzed and stored. However, the process of model checking requires enumerating all possible states or actions of the system to track the ones which do not match the specification. To preserve termination and to improve efficiency, model checking algorithms have to keep track of the visited states in a state space. Therefore, early algorithms based on an explicit reachable state enumeration suffer from the well-known *state space explosion* problem, since the number of states can grow exponentially in the number of variables. Thus, explicit model checking algorithms were only able to handle small designs and did not scale to examples with industrial complexity.

2.2.2.1 Symbolic Model Checking

Symbolic model checking was proposed to avoid the explicit construction of states [27]. In symbolic model checking, sets of states are represented implicitly using Boolean formulas over the set of binary variables used to encode the transition system. For example, we can use Boolean function $v_1 \vee v_2$, instead of explicitly listing 01, 10, 11, to store and manipulate the three states efficiently, where v_1 and v_2 are state variables. *Reduced Ordered Binary Decision Diagram* (ROBDD) [29] was traditionally a data structure that provides a compact, canonical graph representation of propositional formulae. In 1993, McMillan first proposed to use BDDs in symbolic model checking [30], where BDDs serve as implicit representation of states and the operations during model checking process, like reachable state computation, can all be manipulated directly on BDDs. The combination of model checking with BDDs made it possible to explore state spaces of extremely large size, and became a very popular model checking method, especially in hardware verification.

Although BDD-based symbolic model checking offers the potential of handling large state spaces, in practice, it still suffers from the problem of large amount of memory requirement for storing and manipulating BDDs. The Boolean functions required to represent the set of states can still grow exponentially, therefore, full verification of many designs is still beyond the capacity of BDD-based symbolic model checkers.

2.2.2.2 Bounded Model Checking

An alternative method to symbolic model checking is *Bounded Model Checking* (BMC), proposed by Biere et al. in 1999 [1]. The basic idea in BMC is to search for a counterexample in executions whose length is bounded by some integer k. If no bug is found, then one may successively increase k until either a bug is found, or some pre-defined upper bound is reached. The BMC problem can be efficiently reduced to a *Boolean satisfiability problem* (SAT):

Definition 2.1 Boolean Satisfiability Problem. A *variable* can be either true or false. A *literal* is either a variable or the negation of a variable. A *clause* is a disjunction of literals,

and a *Boolean satisfiability problem (in Conjunctive Normal Form (CNF))*, or a SAT problem, is a conjunction of clauses.

A *solution* to a SAT problem *S* is an assignment of values to each of the variables so that every clause in *S* is satisfied. For example, a=1, b=0, c=1 is one of the solutions for a CNF formula: $(a \vee b) \wedge (a' \vee c) \wedge (c' \vee b')$.

The basic process of SAT-based BMC is as follows: Given a transition system *M* encoded as a propositional formula, a property expressed as a temporal logic *P* and a depth limit *k*, it generates a Boolean formula with finite unraveling of the transition relation, which is satisfiable if and only if there exists a witness/counterexample to *P* of length *k* in the system *M*.

Compared to BDDs, the biggest advantage of SAT-based BMC is its space efficiency. The satisfiability check is typically performed by a back-end SAT solver, which does not have the same space explosion problem of BDD-based methods. With the recent advances in SAT solvers [31], SAT-based BMC can handle much larger designs with hundreds of thousands of variables or more in practice. Besides space efficiency, BMC also produces counterexamples of minimal length, which eases their interpretation and understanding for debugging purposes. On the other hand, BMC also has the disadvantage of not being able to prove the absence of errors in some realistic cases. Since BMC provides a bound on the number of iterations that should be explored, which implies that the method is incomplete unless we can determine a high enough value for the bound *k* which guarantees that no counterexample has been missed. Therefore, BMC is usually adopted as a complementary technique to BDD-based symbolic model checking rather than a replacement.

2.3 Software Model Checking

The success and benefits achieved by applying model checking in hardware domains as well as the difficulty of producing correct software are driving a growing interest of gaining a similar success in the context of software verification. However, academic efforts in correctness proof of computer programs tracing back to early days of computer

science have long been ignored by industry as "impractical", and the reason lies in the fact that drastic differences exist between the characteristics of software and hardware systems. Unlike hardware systems, which have strict structures, restricted sizes and are inherently finite in the number of states, software systems have potentially infinite state, more complicated control flow and data structure, richer modularity constructs. They all lead to many challenges of scientific and practical interests, both in core model checking technology and in supporting techniques. While the challenges still remain, some progresses have been made and a variety of successful tools have emerged for automated software verification in recent years, some of them have found applications in the industry. In this section, we'll review some general approaches and recent advances in software model checking.

2.3.1 Abstraction Techniques

Despite the success of symbolic methods, the state explosion problem remains a major hurdle in applying model checking to large industrial designs. A number of approaches targeting state reduction have been proposed to reduce the number of states in the model, including symmetry reductions [79, 80, 81], partial order reductions [82, 83], and abstraction techniques [4, 69, 84]. Among them, abstraction is considered the most general and flexible for handling the state space explosion problem [78]. For software, abstraction becomes even more critical due to the typically infinite-state space.

As the name suggests, *abstraction* defines the activity of removing components and/or simplifying details of the original design that are irrelevant to a certain purpose, which in the context of model checking, refers to the target property checking. Intuitively, verifying the simplified ("abstract") model may lead to performance improvement when compared with verifying the original model. However, the information loss brought by the abstraction may also incur incorrect verification results depending on how the abstraction is conducted. There are two mainstream abstraction techniques distinguished by how the system is simplified:

1) Under-approximation:

Under-approximation [76, 84] is a technique that systematically removes behaviors deemed "irrelevant" from the system, such that the abstract system contains a *subset* of behavior of the original concrete system. Therefore, a counterexample found in the abstract system implies a refutation of property in the original system. However, under-approximation admits false positive, i.e., if the property is found satisfied in the abstract system, it may not necessarily hold in the original system.

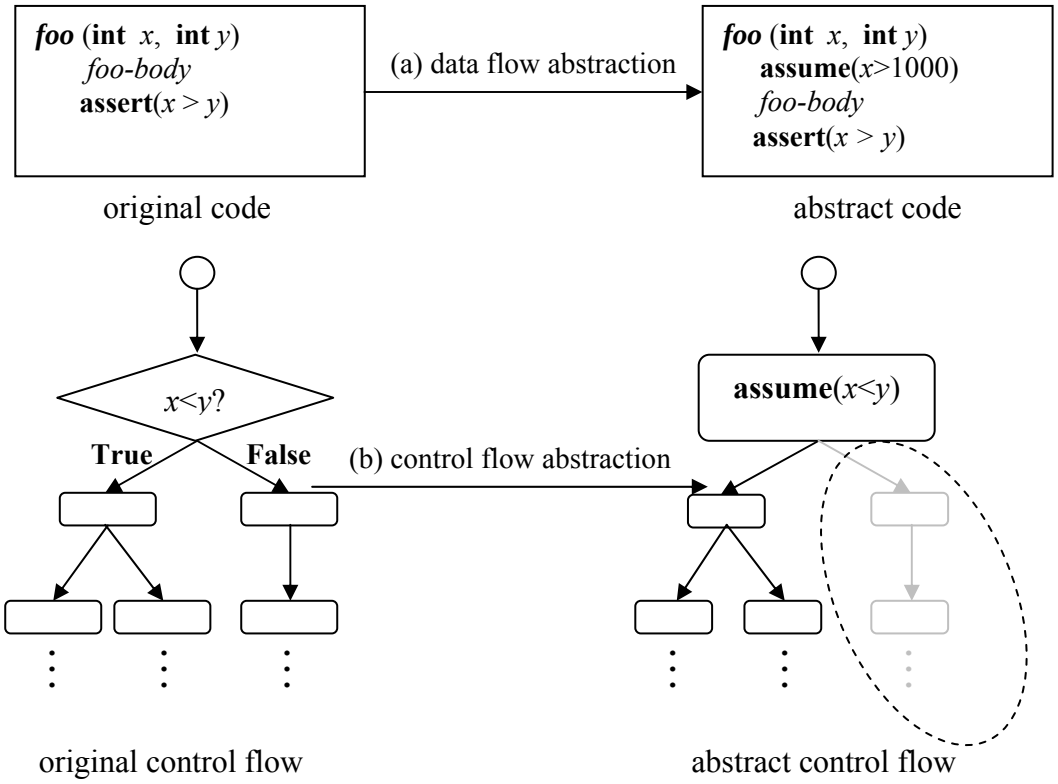


Figure 2.5 Examples for Under-Approximation Based Software Abstraction

In the context of software verification, there are two popular under-approximation based program abstraction techniques: data flow abstraction and control flow abstraction. Data flow abstraction usually reduces the state space by constraining the value range or storage capacity of data domain, for instance, reducing the variable bit-width, shrinking array bounds, etc. Figure 2.5(a) illustrates an example of data abstraction by limiting the input values of variable *x* in the abstract program to a range of (1000, MAX_INT] rather than all the integer values through the ***assume***() statement. In case of control flow abstraction, on the other hand, the program complexity is reduced from the perspective of program

structure slicing. As the example shown in Figure 2.5(b), by assuming that the conditional expression "$x<y$" is always taken, we can obtain an under-approximation of program behavior by cutting out the else-branch within the dotted circle. Compared to the conventional static program slicing, the control flow abstraction does not guarantee that the removed parts indeed have no influence to the target property checking.

2) Over-approximation:

In contrast, over-approximation [85, 86] is a technique that systematically relaxes constraints in the system, such that the abstract system contains a *superset* of behavior of the original concrete system. Therefore, if the property holds in the abstract system, it definitely implies the correctness in the original system. Different from under-approximation, over-approximation admits false negatives, i.e., a counterexample found in the abstract system may not find its concrete counterpart in the original system.

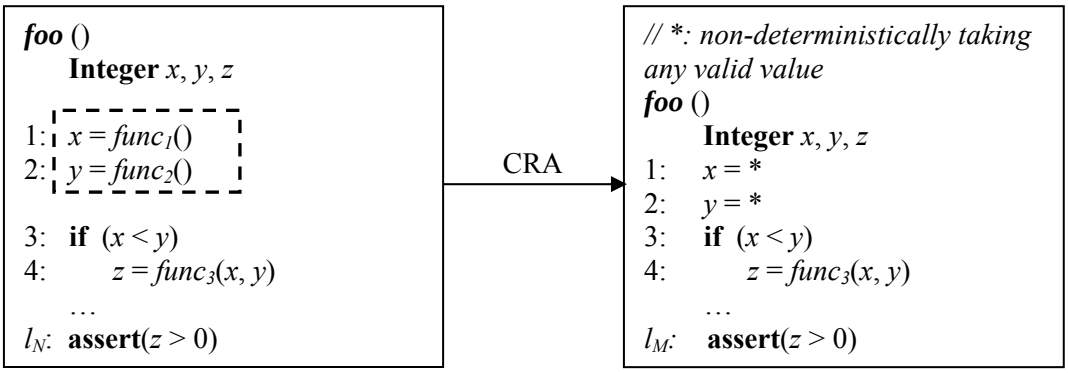
In the next few sections, we'll introduce two specific over-approximate abstraction strategies that have been applied to facilitate software model checking.

2.3.1.1 Constraint-Reduction-Based Abstraction (CRA)

Constraint-reduction-based abstraction (or *localization reduction*, or *variable hiding*) [35, 71, 74, 76] is a category of abstraction techniques that reduces the design complexity by fully relaxing the constraints over selected set of state variables. Given a concrete transition function, CRA partitions the state variables into *invisible* and *visible* sub-sets. The abstract model is created by removing the transition relations over invisible variables and making them as *pseudo-primary inputs*, while preserving the transition relations over visible variables. Since the invisible variables are left unconstrained with free behaviors, the abstract model built by CRA is a conservative over-approximation that keeps all possible execution traces of the original model with probably additional behaviors introduced. CRA has been used commonly in abstracting large hardware designs [35], especially when combined with the use of SAT solvers [87– 91].

A software program can also be viewed as composed of a set of constraints which are defined over high-level variables. Similarly, in the software domain, CRA refers to

releasing the definition of invisible program variables and making them as pseudo program inputs. Figure 2.6 gives an example of program abstraction using CRA. The original concrete program under verification is shown in Figure 2.6(a). For software model checking, the target property that the program is checking against is generally represented using assertion statements injected in the code, for instance, the **assert**($z > 0$) at line l_N in Figure 2.6(a). The resulting abstract program is shown in Figure 2.6(b). With CRA, local variables − integers x and y, originally obtained their values from $func_1()$ and $func_2()$, respectively, are freed in the abstract program, which means they can now take any arbitrary integer values. The two freed variables are treated as if the function *foo* gains two more pseudo input parameters.



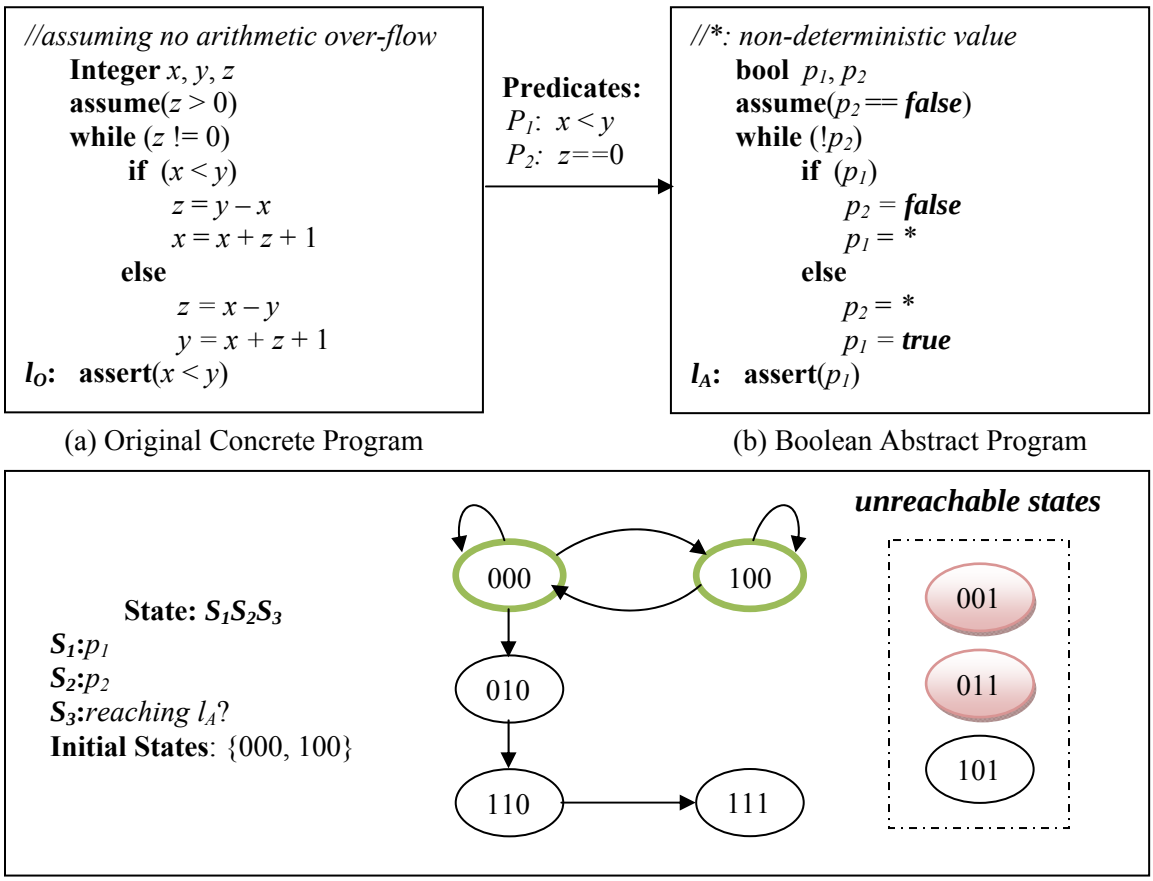
(a) Original Concrete Program (b) CRA Abstract Program

Figure 2.6 Constraint-Reduction-Based Abstraction (CRA) Example

The abstraction computation of CRA is quite efficient since the abstract model can be built directly given a set of visible variables.

2.3.1.2 Predicate Abstraction

Predicate abstraction, first proposed by Graf and Saidi [69], is a promising over-approximation based abstraction technique. It can automatically construct an abstract finite state system of arbitrary infinite state systems where the abstract states are valuations of a set of predicates defined over concrete variables. Therefore, it becomes one of the most popular and widely applied methods for systematic abstraction of designs with potentially infinite size, e.g., programs.



(a) Original Concrete Program (b) Boolean Abstract Program

(c) Finite State System

Figure 2.7 Predicate Abstraction Example

In the context of software model checking [26, 34, 73], predicate abstraction is performed by characterizing a program in terms of how it transforms the truth values of a finite set of predicates over program variables. Take Figure 2.7 as an example, given the original concrete program in Figure 2.7(a) and the target property to be checked **assert**(x<y) at line l_O, suppose two predicates {P_1: x<y, P_2: z==0} are used to construct abstraction, where x, y and z are original program variables, the resulting Boolean abstract program is shown in Figure 2.7(b). Each predicate is represented by a Boolean variable in the abstract program, e.g., ***bool*** p_1 for P_1. Note that the **assume**(ψ) statement implies that the Boolean expression ψ is evaluated to true at the location. With predicate abstraction, the abstract model obtained is a finite state machine model containing only these Boolean variables while the original data variables are eliminated. In Figure 2.7(b), the concrete states {x, y, z} of the original program are mapped to abstract states {p_1, p_2} whose values are evaluated under the definitions of predicates P_1 and P_2. The Boolean abstract program

can be modeled using a finite state machine (FSM) in Figure 2.7(c). Each state in the FSM is composed of three state variables: $s_1s_2s_3$, representing Boolean variables p_1, p_2, and whether the system reaches the line l_A where the assertion is located, respectively.

The problem of model checking the *assertion(x<y)* in the original concrete program now has been transformed into the proving of the un-reachability of two error states: {001, 011} in the abstract finite state system. Predicate abstraction is conservative in the sense that for every execution in the concrete system there is a corresponding execution in the abstract system, but may not be the case, vice versa. According to the Figure 2.7(c), starting from the two initial states: {000, 100}, the error states are NOT reachable, thus, the assertion holds in the original concrete program in Figure 2.7(a). Two major ways of constructing the Boolean abstract program are: through theorem proving [34, 73] or SAT-solvers [26].

2.3.2 Counter-Example Guided Abstraction Refinement (CEGAR)

Over the past few years, automated iterative abstraction refinement scheme, usually coupled with abstraction techniques (e.g., predicate abstraction), has become a popular approach for software model checking. Examples of model checkers that implemented this scheme include SLAM [33], BLAST [34], MAGIC [36], etc. Generally, the process contains a loop of the following three basic steps: *model construction*, *model checking* and *abstraction refinement*. This three-step-loop is also called Counter-Example Guided Abstraction Refinement (CEGAR) [35, 76] paradigm. The basic work flow is shown in Figure 2.8. CEGAR is first used to model check software programs by Ball and Rajamani in [77].

The purpose of the first step of CEGAR – model construction is to extract a finite state abstract model from the infinite state software system using the selected abstraction technique. For instance, the Figure 2.7 above shows an example of first step with predicate abstraction.

At the second step, the abstracted system is model checked to see whether it satisfies the target property. If it succeeds, the property holds in the concrete system and verification

process terminates; otherwise, an abstract counterexample trace is generated. Since the abstracted system is created by a conservative approximation, the abstract counterexample may have two possible cases. There could be a concrete counterexample trace (or actual bug) corresponding to the abstract one or it may happen that the abstract counterexample is a product of abstraction and not feasible in the concrete system. The latter case is called *spurious*.

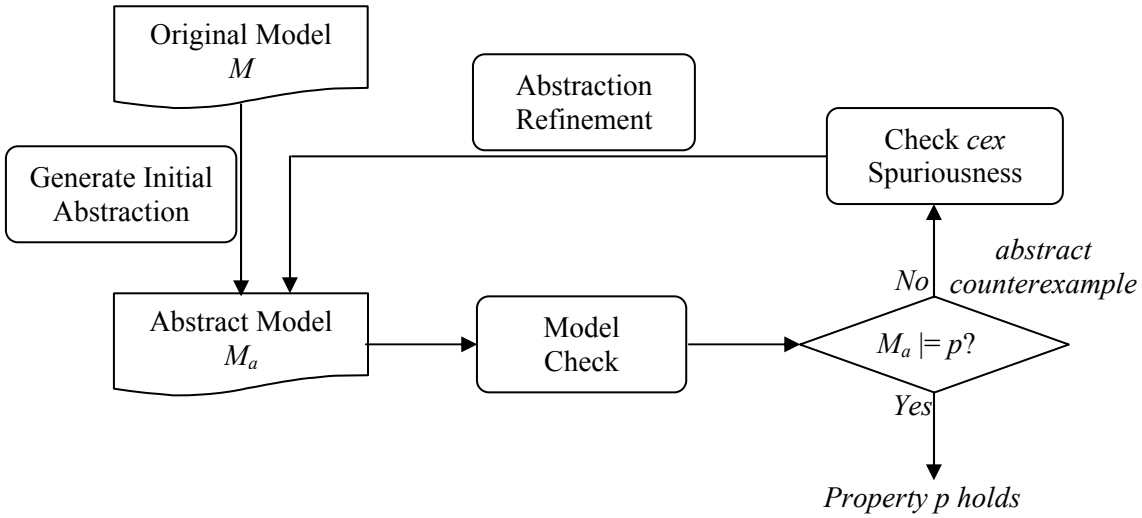


Figure 2.8 CEGAR Paradigm

Once a counterexample is verified to be spurious, an abstraction refinement process is applied to gradually restore details into the abstract model until the system is sufficient to prove or refute the property. For example, in the case of predicate abstraction, the refinement process will add to the set of predicates with new predicates such that the previous spurious counterexamples get eliminated if the model is abstracted using the new predicate set. A finer abstraction is then created with the updated predicate set, and continues with model checking until either the property is proved or a concrete counterexample is found.

2.3.3 SAT-Based Software Bounded Model Checking

Instead of using predicate abstraction, bounded model checking is another way to extract a finite state model out of an inherently infinite state software system. In this approach, the software code is transformed into a bit-level propositional formula by unwinding the

potentially infinite constructs (e.g., **while** loops) for only a fixed number of times. A representative example of software bounded model checker for ANSI-C programs is CBMC [6].

The basic process of CBMC is as follows: given a C program, property P to be checked, and a pre-defined unwinding depth *N*, before the bit-level translation, a set of pre-processing transformations need to be performed at the high-level (Figure 2.9), which includes:

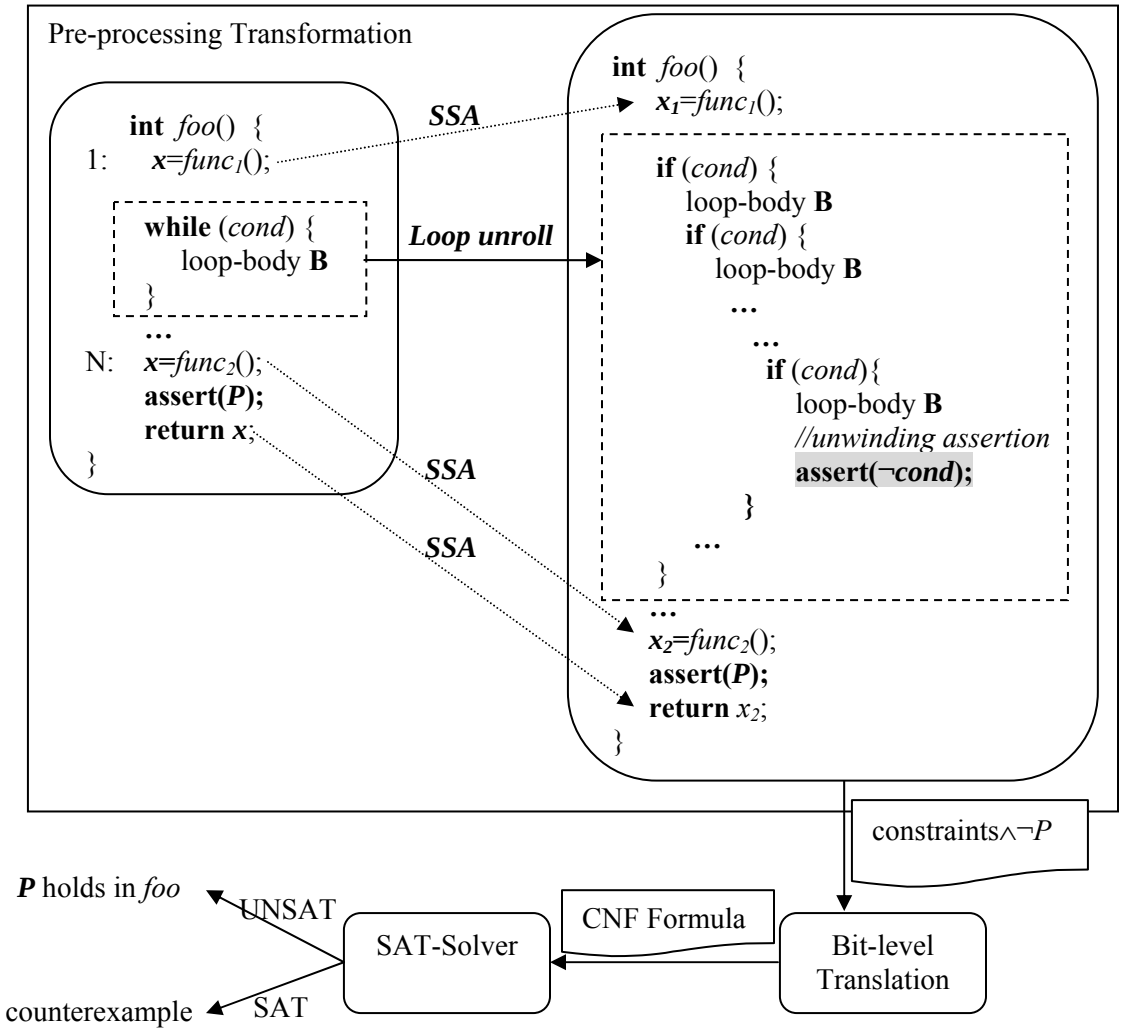


Figure 2.9 Software Bounded Model Checking (CBMC)

(1) Inserting the specified property *P* to be checked to the program as assertions;

(2) Eliminating loops by unwinding loops for N times via duplicating the loop body, at the same time, inserting an *unwinding assertion* using negated loop condition at the end of the N repetitions to assert that the unwinding bound N is actually large enough. Figure 2.9 shows an example of unrolling the **while** loop using a series of nested **if** statements, and "**assert**(¬*cond*)" is appended to the end of the last **if** statement, which is violated if the program runs longer than the bound permits. Recursions are treated in a similar way to loops;

(3) Converting the input program to *Static Single Assignment* (SSA) [104] form where each variable is assigned only once. For example, in Figure 2.9, the variable x is defined twice at line 1 and line N of the original code, respectively, in the SSA form, we rename the two variables into x_1 and x_2, such that no variable is assigned multiple times. The SSA form helps ease the complexity of the subsequent bit-level translation.

After the above preprocessing transformation, all the statements in the resulting program can be viewed as a conjunction of constraints C. CBMC then converts the $C \wedge \neg P$ into a CNF formula by representing each variable as a bit vector with width determined by the data type of the variable and the conversion of most operators resembles the generation of appropriate arithmetic circuits. Finally, SAT-solver is utilized to solve the CNF like in other bounded model checkers.

2.4 Invariant Discovery Techniques

The invariants in a program represent the properties of program behaviors that constantly hold during the program execution. In this section, we introduce and compare two invariant extraction techniques: static program analysis and data mining.

2.4.1 Static Program Analysis

Static program analysis [4, 13] is a technique that analyzes the code for gathering the properties of the run-time behavior of a program without actually executing it. It has been widely applied in the area of compiler optimization, program transformation, etc. Some of the well-known static analysis examples include: data flow analysis, interval analysis,

dependency analysis and so on.

Generally, the process of static program analysis contains two phases. The first phase performs an *abstract interpretation* of the program to formally model a sound approximation of the program semantics. As a result, a system of equations or constraints is constructed involving a set of monotone functions over lattices. A lattice (S, ⊑) is a partially ordered set that defines a binary relation ⊑ over set S which is reflexive, anti-symmetric, and transitive, and for all X ⊑ S, there exist a least upper bound and a greatest lower bound. For further information about the lattice theory, we refer the readers to [106]. The solution of these equations represents the target invariance properties of the program to be found by the analysis. The second phase aims at finding such a solution through a *fixpoint iteration* algorithm.

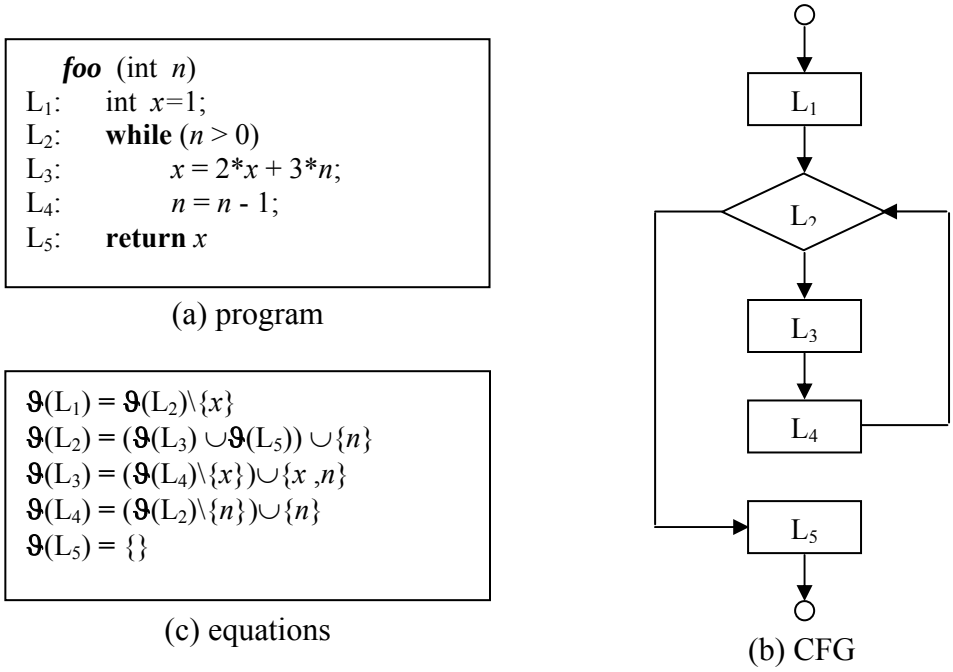


Figure 2.10 An Example for Live Variable Analysis

Figure 2.10 shows an example of using static analysis technique to compute live variables for each program location. A variable is *live* at a program point (i.e., the entry of a program point) if it may be read in the remaining program execution before its next definition. Given the program in Figure 2.10(a), the domain of live variables can thus be modeled by the lattice ($2^{\{x, n\}}$, ⊑). Correspondingly, we can construct the control flow

graph (CFG) of the program as shown in Figure 2.10(b). Let $\mathfrak{I}(L)$ denote the set of live variables at the entry point of a program location L, it can be computed using the following equations:

$$\begin{aligned} \mathfrak{I}(L) &= \left(Join\,(L) \setminus Kill\,(L)\right) \cup Gen\,(L)\right) \\ Join\,(L) &= \bigcup_{S \in Succ\,(L)} \mathfrak{I}(S) \end{aligned} \qquad (2.1)$$

where $Kill(L)$ represents the set of variables defined in the statement at L, $Gen(L)$ represents the set of variables used before a re-definition in the statement at L, and $Join(L)$ denotes the union of the live variable sets from all the potential program points located successive to L according to the CFG. For example, $Join(L_1)= \mathfrak{I}(L_2)$, $Join(L_2)=(\mathfrak{I}(L_3) \cup \mathfrak{I}(L_5))$. Then, using the rules in (2.1), we can obtain a set of monotone equations over the lattice $(2^{\{x, n\}}, \sqsubseteq)$ for each program point, shown in Figure 2.10(c).

In order to solve the equations, a fixpoint computation algorithm is applied to iteratively walk up the lattice, starting from {}, until reaching a fixpoint at all the program locations. That is, by treating each of the right-hand side of the equations in Figure 2.10(c) as a function Fun_i ($1 \le i \le 5$), a fixpoint solution x satisfies that $Fun_i(x) = x$. As shown in Figure 2.11(a), we start from {} at every program point, then, iteratively compute the new value of $\mathfrak{I}(L_i)$ for every L_i using the results of $\mathfrak{I}(L_i)$ ($1 \le i \le 5$) from last iteration according to the equations in Figure 2.10(c). We reach a fixpoint at the 4th iteration (Figure 2.11(d)), since the 5th iteration will generate exactly the same results of $\mathfrak{I}(L_i)$ as in the 4th iteration. The results in Figure 2.11(d) are the solutions for live variables.

(a) 1st iteration	(b) 2nd iteration	(c) 3rd iteration	(d) fix-point
$\mathfrak{I}(L_1) = \{\}$	$\mathfrak{I}(L_1) = \{\}$	$\mathfrak{I}(L_1) = \{n\}$	$\mathfrak{I}(L_1) = \{n\}$
$\mathfrak{I}(L_2) = \{\}$	$\mathfrak{I}(L_2) = \{n\}$	$\mathfrak{I}(L_2) = \{x, n\}$	$\mathfrak{I}(L_2) = \{x, n\}$
$\mathfrak{I}(L_3) = \{\}$	$\mathfrak{I}(L_3) = \{x, n\}$	$\mathfrak{I}(L_3) = \{x, n\}$	$\mathfrak{I}(L_3) = \{x, n\}$
$\mathfrak{I}(L_4) = \{\}$	$\mathfrak{I}(L_4) = \{n\}$	$\mathfrak{I}(L_4) = \{n\}$	$\mathfrak{I}(L_4) = \{x, n\}$
$\mathfrak{I}(L_5) = \{\}$	$\mathfrak{I}(L_5) = \{\}$	$\mathfrak{I}(L_5) = \{\}$	$\mathfrak{I}(L_5) = \{\}$

Figure 2.11 Fix-Point Computation

2.4.2 Data Mining

Data mining is an analytic procedure designed to explore data (usually large amounts of data) from different perspectives in search of consistent patterns and/or systematic relationships between variables. In this section, we briefly introduce the background knowledge of a specific data mining technique — association rule mining.

2.4.2.1 Basic Definition of Association Rule

Association rule mining provides a useful mechanism for discovering rules such as implication or correlations among items that co-occur frequently within a data set.

A data set usually consists of M independent *transactions*, each of which is a set of N_i *items* drawn from a set of all possible N items. For example, Table 2.1 gives a tabula format of a data set consisting of 4 transactions with 5 items {*I1*, *I2*, *I3*, *I4*, *I5*}, where the symbol '×' means the occurrence of an item in a given transaction.

Table 2.1 An Example Data Set for Association Rule Mining

Transaction	*I1*	*I2*	*I3*	*I4*	*I5*
1	×		×	×	
2		×	×		×
3	×		×	×	
4				×	×

An association rule has the form of $A \Rightarrow B$, where A and B are sets of items or itemsets and $A \cap B = \varnothing$. A k-itemset is a set containing k items. The strength of an association rule is usually measured by two metrics: *support* and *confidence*. The support is a fraction of transactions in the data set that contain both A and B:

$$support(A \Rightarrow B) = P(A \cap B) = Count(A \cap B)/M$$

where the Count(x) represents the total number of transactions that satisfies condition x. The confidence of an association rule is a measure of the reliability and accuracy of the rule.

$$confidence(A \Rightarrow B) = P(B \mid A) = \text{Count}(A \cap B)/\text{Count}(A)$$

The association rules problem is that of computing all association rules that satisfy user-specified minimum support and minimum confidence threshold.

2.4.2.2 The A-Priori Algorithm

A-Priori is a classic algorithm for mining association rules. It consists of two major steps: The first step is frequent itemset generation by employing a *level-wise* search. In particular, the computed frequent k-itemsets are used to find the frequent (k+1)-itemsets. Take the data set in Table 2.1 as an example, suppose the support threshold s_{th} = 0.5, starting with iteration k=1, we first compute the support value of each possible 1-itemset and those 1-itemsets whose support reach s_{th} generate the frequent 1-itemset L_1={*I1*, *I3*, *I4*, *I5*} as shown in Table 2.2.

Table 2.2 Frequent 1-Itemset (s_{th}=0.5)

Candidate 1-itemsets	Count	Support	Frequent 1-itemset ?
{I1}	2	2/4=0.5	Y
{I2}	1	1/4=0.25	N
{I3}	3	3/4=0.75	Y
{I4}	3	3/4=0.75	Y
{I5}	2	2/4=0.5	Y

Table 2.3 Frequent 2-Itemset (s_{th}=0.5)

Candidate 2-itemsets	Count	Support	Frequent 2-itemset ?
{I1, I3}	2	2/4=0.5	Y
{I1, I4}	2	2/4=0.5	Y
{I1, I5}	0	0/4=0	N
{I3, I4}	2	2/4=0.5	Y
{I3, I5}	1	1/4=0.25	N
{I4, I5}	1	1/4=0.25	N

Next, at the second iteration, we construct a candidate 2-itemsets C_2 by listing all pairs of

itemsets from the frequent 1-itemset L_1. Then, the data set in Table 2.1 is scanned again to count the occurrence of each candidate 2-itemset in C_2 as shown in Table 2.3. The results are finally pruned using support threshold s_{th} and form the frequent 2-itemset L_2 = {{*I1*, *I3*}, {*I1*, *I4*}, {*I3*, *I4*}}. The above process can be repeated until iteration k reaches certain pre-defined value or the candidate sets become empty.

The second step of the A-Priori algorithm is the association rule derivation. Suppose the frequent itemset generation stops at iteration k, for each frequent k-itemset f in L_k, all of its non-empty proper subsets can be enumerated. Then for each subset of f denoted as f_s, a set of candidate association rule in the format of R: $f_s \Rightarrow (f - f_s)$ is generated, and output R if it fulfills the minimum confidence requirement. For example, for frequent 2-itemset {*I1*, *I4*}, all of its non-empty proper subsets are {*I1*} and {*I4*}, we can thus generate two candidate association rules: *I1*⇒*I4* and *I4*⇒*I1*. Based on the earlier definition of the confidence, *confidence*(*I1*⇒*I4*)=2/2=1, *confidence*(*I4*⇒*I1*)=2/3≈0.67. Suppose the confidence threshold c_{th} is set to 0.8, then *I1*⇒*I4* is an association rule for frequent 2-itemset {*I1*, *I4*}.

Table 2.4 Association Rule Derivation (c_{th}=0.8)

Candidate Association Rules	Support	Confidence	Association Rule?
I1 ⇒ I4	2/4=0.5	2/2=1	Y
I4 ⇒ I1	2/4=0.5	2/3≈0.67	N
I1 ⇒ I3	2/4=0.5	2/2=1.0	Y
I3 ⇒ I1	2/4=0.5	2/3≈0.67	N
I4 ⇒ I3	2/4=0.5	2/3≈0.67	N
I3 ⇒ I4	2/4=0.5	2/3≈0.67	N

Table 2.4 lists all the candidate association rules and the corresponding support and confidence values for the above frequent 2-itemsets L_2. Given c_{th} = 0.8, the resulting association rules are *I1*⇒*I4* and *I1*⇒*I3*.

Compared to the conventional static program analysis approaches for constraints learning, data mining has the following advantages:

1) Straightforward. In contrast to static analysis, where approximations about the possible set of values at various points in a program are computed without actually executing the code, data mining is directly based on the profile data collected during actual program execution. In addition, it requires the resulting property to be unconditionally true over the simulated data set. Therefore, data mining can directly reflect the actual program control flow via more straightforward profiling and learning compared to the conservative feature in static program analysis.

2) Flexible. The data mining techniques can be easily utilized in learning almost any formats of property without modifying the data set repository being mined. However, in static analysis, different models may need to be constructed for each different target property format.

2.4.3 Invariant Extraction and Model Checking

A large amount of research work has been done in the area of model checking and data mining separately, but not much was engaged in the combination of both. This may be probably due to the somewhat conflict features in correctness requirements between data mining and verification. In terms of mining, a machine learning approach was used in [18] to mine formal specifications by observing the program executions, assuming the program is generally correct. For hardware, in [5], data mining was applied for global constraints mining to improve the performance of SAT-based bounded equivalence checking of sequential circuits.

Some previous work was proposed to assist software verification by adding property invariants. Most of these properties are learned using static analysis. Two different types of learning were applied: offline and online. In the offline case, invariants are computed relatively independent from verification process. Jain et al. added statically computed invariants in the form of $\pm x \pm y \leq c$ at given program location to strengthen the concrete transition relation [100] and thus helped reduce the total number of abstraction refinement iterations and the overall verification time for predicate abstraction. Extracting high-level design information from EFSM model to accelerate high-level BMC is one of the mechanisms proposed in [3], where most of these invariants are just

explicit control flow information between the two adjacent basic blocks. In the online case, invariants are deduced and strengthened during the verification process iteratively to help reduce the search space [8, 9, 14, 15].

The offline approaches only need to run once but suffer the problems of effectiveness since the invariants learned may not fit the verification requirements of an assertion. While the online approach has the advantage of deducing appropriate invariants when it needs to but requires being conducted iteratively which might incur extra cost.

Besides static analysis, there are also other invariants discovery techniques proposed in the literature. Ernst et al. in [16, 17] proposed a dynamic program execution based method in tool Daikon for likely program property and conditional implication learning. The learned properties can be sent to static checker for correctness verification and were applied to specification inference, error-detection, etc., but none of them are used as additional knowledge to improve performance of model checking.

2.5 Swarm Intelligence

Swarm Intelligence (SI) is a discipline that involves the study of natural and artificial systems typically composed of a population of simple individuals coordinating with each other and with their environment using decentralized control and self-organization. In particular, the discipline focuses on the collective behaviors that result from the local interactions among individuals. These interactions often cause an "intelligent", global pattern to emerge, although each individual follows very simple rule and no centralized control structure for individual behavior dictation is available [56]. The expression was first introduced by Beni and Wang [55] in the context of cellular robotic systems made up by many simple agents through nearest-neighbor interactions. Natural examples of systems studied by swarm intelligence include ant colonies, animal herding, bacteria foraging, bird flocking, fish schooling, honey bees, and many more, which lead to various scientific and engineering swarm intelligence algorithms, such as Stochastic Diffusion Search (SDS), Particle Swarm Optimization (PSO), etc. In the following section, we will

introduce one of the most successful examples of artificial/engineering SI system – Ant Colony Optimization (ACO), in more detail.

2.5.1 Ant Colony Optimization

Ant colony optimization ([57], [75]) is a population-based meta-heuristic that can be used to find approximate solutions to difficult optimization problems. It is inspired by the foraging behavior of ant colonies. Initially, a colony of ants wanders randomly in search of food. Once a food source is found, they lay down pheromones along trails with appropriate amounts based on the attractiveness (i.e., quality and quantity) of the food while they return back to the colony. Such pheromone trails thus establish indirect communication links among ants and provide guidance for the future ants towards the food source along the most likely best (shortest) paths. Pheromones get accumulated on different paths, as well as evaporate over time at a rate proportional to the length of the path between the food and the colony.

Similarly, in the ACO algorithm, a set of software agents called "artificial ants" search for good solutions to a given optimization problem, which is usually transformed into the problem of finding the minimum cost path on a weighted graph. The general framework of the ACO algorithm is as follows:

	Algorithm Ant-Colony-Optimization**()**
1:	Initialization()
2:	while (!end_condition)
3:	constructSolution()
4:	updatePheromones()
5:	daemonActions() {optional}

Figure 2.12 General ACO Algorithm

Initially, the artificial ants move around the graph iteratively trying to find possible solutions (e.g., paths in the graph). In each iteration, ants construct possible solutions stochastically and biased by a pheromone model: the edge leading to the next node with a higher pheromone level is more likely to be followed. Here the pheromone indicates a set

of parameters associated with graph components (either nodes or edges), the values of which are modified at runtime by the ants. Then, a heuristic measure is used to evaluate the "quality" of the solution and update the pheromone intensity accordingly along the solution path. The optional daemon actions are usually used to implement centralized mechanisms which cannot be performed by single ants. Generally, the purpose of such actions is to decide whether it is useful to bias the search process from a non-local perspective by tuning current pheromone solutions built by the ants. One application of daemon is to prevent the searching from being trapped in the "local optimum". The above process repeats until the terminating conditions are met.

Ant System [57] was the first known ACO algorithm that applied artificial ant colonies to solve the traveling salesman problem. Since then, many variants have been applied to solve various classical combinatorial optimization problems, as well as to discrete optimization problems that have stochastic and/or dynamic components. In recent years, the ACO has been utilized for test input generation in various contexts: from Markov software usage model [58], based on UML state chart diagrams [60], for event sequence construction [59] and for mutation testing [61]. However, none of the previous work has been conducted to associate ACO with program abstraction. By reexamining the ACO algorithm, we find that it tends to identify high-quality paths where the solutions most likely locate through the cooperation among ant agents. When sending the ants to walk through the program structure, it provides an effective means to distinguish promising program areas out of less valuable ones with respect to the goal – target property checking, thus, can be utilized to guide the program abstraction through code slicing. In Chapter 5, we'll present the new ACO-directed under-approximation-based program abstraction technique.

Chapter 3

Internal Variable Range Coverage Metric and Test Generation Model

In this chapter, we present new Internal Variable Range (IVR) coverage metric for sub-domain based partition testing. It uses a dynamic-program-execution-based approach to project the input domains onto approximate ranges of critical internal variables, and then partition the internal variable domain based on the data value distribution as well as the program structure information related to the internal variables. A sub-range transition system is also proposed to facilitate the internal variable range coverage oriented automatic test data generation process. Our experimental results on a class of extended triangle programs showed that our model can generate test set that achieves much higher internal variable range coverage than random test set. And the new coverage metric also provides an effective estimation of the fault detection capability of an input test set, especially for programs that are arithmetic computation related.

The remainder of the chapter is organized as follows. Section 3.1 gives a motivating example to explain our idea. In Section 3.2, we introduce the algorithm of dynamic-program-execution-based internal variable range partition and the corresponding coverage metric. Our finite state model of automated test generation to achieve the range coverage is presented in Section 3.3. In Section 3.4, we use a class of programs that extends the classical triangle program as benchmarks to investigate the fault detection

capability of the test generation based on our criteria. Finally, Section 3.5 concludes the contribution of this work.

3.1 A Motivating Example

As shown by the example code segment in Figure 3.1, suppose $X_1, \ldots, X_n$ are n input variables to the function *foo*, $V_1, \ldots, V_m$ are m internal variables that are used to store intermediate logical and/or computational results. Each internal variable can be regarded as a function of all the previously defined variables. Generally, the internal variables $V_1, \ldots, V_m$ would be more likely to be involved in the subsequent operations of the function than the external variables, like V_i ($1 \leq i \leq m$) in the if-then-else statement shown in Figure 3.1. Thus, partitioning of V_i's would provide a more direct guidance to the testing of the program.

```
Function foo (X1, ..., Xn)
        V1 = f1 (X1, ..., Xn);
        V2 = f2 (X1, ..., Xn, V1);
        …
        Vm = fm (X1, ..., Xn, V1, ..., Vm-1);
        …
        if (Vi op ConstantA)
            then …
            else …
        …
```

Figure 3.1 A Motivating Code Example for Range Partitioning

In addition, by selecting critical internal variables and partition the value ranges for each of them separately, it can transform the high-dimension input space into a lower dimension internal variable space, which is more manageable in terms of partition. Take Figure 3.1 as an example, suppose V_i is the selected critical internal variable, the sub-domain partition then is reduced from n-dimension to 1-dimension, and since V_i is a function of $X_1, \ldots, X_n$, any partitions on $X_1, \ldots, X_n$ can still be projected onto V_i without losing important information. In other words, the internal variable partition adds an intermediate layer between the conventional input-output mappings of the input domain

partition testing, which simplifies the process of boundary definition and provides partitions at a more refined granularity.

3.2 Internal Variable Partition Testing

Based on the above analysis, we proposed a new testing strategy to explore internal variable partitioning. It uses a dynamic-program-execution-based approach to project the input domains onto approximate ranges of critical internal variables, and then partition the internal variable domain based on the data value distribution as well as the program structure information related to the internal variables.

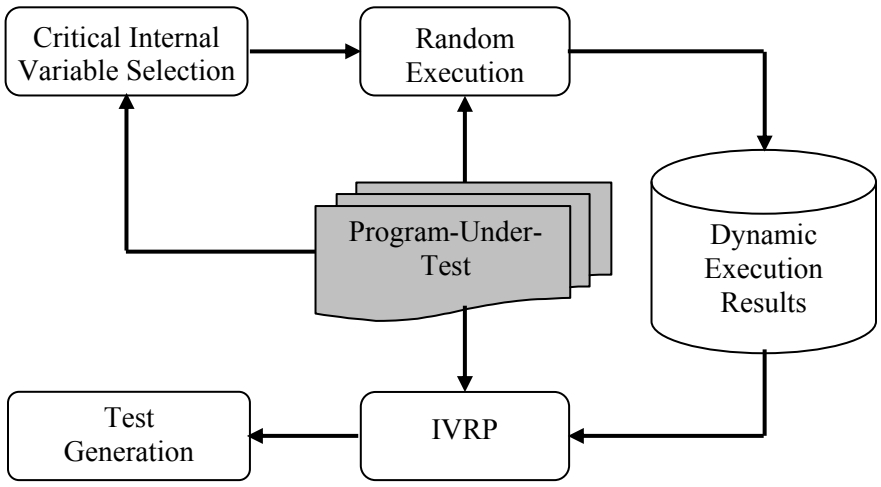


Figure 3.2 Framework of Internal Variable Partition Testing

The framework of Internal Variable Partition (IVP) testing is illustrated in Figure 3.2. It contains three main modules. Given a program under test (PUT), we first analyze the program structure and identify internal variables that are critical. Here, we assume the program implementation is available, so IVP is utilized as a white-box testing technique. Next, we perform random execution on the program and the result data of execution together with the program structure information is fed into Internal Variable Range Partition (IVRP) module for analysis. The main function of IVRP module is composed of two steps: range definition and range merging. Finally, the result of range partition in turn defines the internal variable range coverage metric for the given PUT and can be applied on any automated test generation model.

Code coverage is an important metric to assure the quality of test set in software fault detection. Unlike the common program structure-oriented coverage metric, such as statement coverage, decision coverage, etc, we associate our internal variable partition testing with a new data-oriented coverage metric, called Internal Variable Range (IVR) coverage, which focuses on generating test cases to cover the possible value range combinations of the critical internal variables in a program.

3.2.1 Critical Internal Variable Selection

Internal variables are used as intermediate storages during the program execution. However, there may exist significantly more internal variables than external input variables. Therefore, in our IVP testing strategy, only critical internal variables are involved in the range partition. By defining an internal variable as *critical* indicates that the variable's value has essential influence on the program output or the programmer has specific concern on the correctness of this variable. In this work, we assume that the programmer has the knowledge of the criticality of each internal variable and has the responsibility to make the selection. In order to deal with the situations when internal variables are defined for multiple times at different locations of the program, we assume that the program-under-test has already been transformed into SSA (Static Single Assignment) [104] form. In the following sections, all the internal variables refer to critical internal variables if not mentioned otherwise.

3.2.2 Dynamic-Program-Execution-Based Variable Range Definition

Before partitioning a variable's domain space, either external or internal, we need to determine the possible value range for it. Although the data type of the variable e.g., *int*, *char*, etc., already implicitly defines a boundary for the values that it can contain, it is still too coarse for performing the partition, especially for internal variables. Since internal variables usually represent the intermediate results of some program behavior, instead of uniformly distributed over the entire range, it is highly possible to observe some biased distribution on the actual values of certain internal variables.

In order to obtain the actual data value distribution of critical internal variables, we first apply random execution on the program-under-test and record the values of the internal variables during the running. Given a software program *P* represented in the SSA form, *N* program executions on randomly generated input vectors are conducted and the values for each critical internal variable are recorded. Suppose the program contains *m* critical internal variables ($V_1,\ldots,V_m$), for any given $V_i(1 \leq i \leq m)$, we can obtain an approximate value range [min_i, max_i] based on the random execution results, where min_i is the minimum value among all the *N* simulation results for V_i and max_i is the maximum. Due to the random nature of the execution, we have enough confidence to take these ranges as a close approximation to the real value distribution of the internal variables.

3.2.3 Range Partition and Range Merge

After we obtained the approximate value range for each critical internal variable, the next step is the range partition. An effective range partition will directly influence the quality of the test set generated. In IVP testing, we define the equivalent class based on the data value distribution within each internal variable, by assuming that the internal variable vectors (as opposed to input vectors) which are spatially close to each other in the internal variable domain space would behave similarly on fault detection, especially on data-related errors. At the same time, program structure information is also used in defining range partition such that IVP has the advantage of integrating the strength of both the data- and structure-oriented testing strategies.

A heuristic range partitioning algorithm is proposed. Initially, for each critical internal variable V_i, the simulated range [min_i, max_i] is divided into disjoint sub-ranges with equal distance *d* except at the upper boundary. The distance *d* is a heuristic value depending on at how fine of a granularity one wishes the partition to be. In order to make the partition completely expand over the full domain of V_i, we automatically include the specific language domain limits defined by the variable's data type as sub-range boundaries. For instance, if variable V_i is an integer, [INT_MIN, min_i) and (max_i, INT_MAX] would be appended as two sub-ranges. In addition, program structure information, such as predicates, conditions, that relates directly to the internal variables is also utilized as

predefined sub-range boundaries. For example, if statement "***if*** (V_i ***op*** *constantA*)" exists, then *constantA* will also be inserted as a sub-range boundary.

Besides value ranges, the execution results also reveal a rough distribution of each internal variable. Thus, after the initial range partition, for each sub-range of V_i, we count and record the number of occurrences that the value of V_i falls into the boundary of this sub-range during the N executions. Then, range merging is performed to merge any sub-range that has very few frequencies of occurrences into its neighboring sub-range, so that the values of a given variable are distributed more uniformly among sub-ranges. As in the previous example, the two sub-ranges [INT_MIN, min_i) and (max_i, INT_MAX] will definitely be merged with their corresponding nearby sub-ranges, because the simulation results showed that 0 instances fall into any of the two sub-ranges. A threshold *CNT_TH* is used to guide the range merging, which can be customized accordingly based on tradeoffs between the testing precision and budget. The smaller the value of *CNT_TH*, the more sub-ranges would be partitioned for each internal variable, thus, more test cases need to be generated.

3.2.4 Internal Variable Range Coverage Metric

Based on the internal variable range partition, we can define the IVR coverage metric for the program. Given a program with m critical internal variables V_1, …, V_m, and according to the dynamic-program-execution-based range partition described above, each internal variable V_i is partitioned into C_i sub-ranges: R_{i1}, …, R_{iCi}. Each sub-range R_{ik} is bounded by an interval of [$lowbnd_{ik}$, $upbnd_{ik}$), where $1 \leq k \leq C_i$, $lowbnd_{i1}$ equals to the lower limit defined by V_i's data type and $upbnd_{iCi}$ equals to its upper limit. Thus, the entire domain of V_i is partitioned into mutually exclusive and collectively exhaustive sub-domains. We also define the concept of *sub-range combination* as a m-vector of range index $<r_1, \ldots, r_m>$, where $1 \leq r_i \leq C_i$. The total number of available sub-range combinations under the current range partition is the product of all C_i's ($1 \leq i \leq m$). For example, suppose there are three critical internal variables: V_1, V_2 and V_3 in the program, and they've been partitioned into 10, 4 and 7 sub-ranges, respectively, after conducting the dynamic-program-execution-based range partitioning, then the whole internal variable domain

space would contain 10*4*7=280 sub-range combinations. The IVR coverage measures the coverage of all the internal variable sub-range combinations, that is, for each sub-range combination $<r_1, \ldots, r_m>$, whether one of the test case in the test suite can make the value of each internal variable V_i falls in its corresponding sub-range R_{ri} through a test run on the program.

3.3 Automated Test Generation Model

Two important criteria to evaluate the effectiveness of a code coverage metric are: convenience in test case generation and capability in fault detection. As we know that test generation is an expensive activity in the process of software development, how to reduce the cost by automating the process becomes critical. Much research work has been done in the area of automated test data generation, model-based approaches [102, 103] for functional testing and search-based approaches [38, 101] for structural testing are the most widely utilized techniques. However, according to the specific characteristics shown by our new internal variable range coverage metric, we propose a new dynamic-program-execution-based automated test data generation model: *Sub-Range Transition System* (SRTS), with the goal of covering the internal variable sub-range combination coverage. Compared to the conventional search-based test generation techniques, our SRTS model has the following features:

- It integrates finite state automaton with source-code level test coverage measurement which takes advantage of the benefits in both white-box and model-based testing;
- It builds up a set of sub-models: *Sub-Range Transition*(SRTs) based on information extracted from the random execution results;
- Multiple sub-SRTs work collectively together to achieve the goal state;
- It tunes multiple input variables at the same time during the goal-oriented search process to improve the efficiency of searching.

We will give the detailed illustration on the structure and operations of the proposed model in the following sections.

3.3.1 Sub-Range Transition System (SRTS)

Once the range analysis and partition for a given program have been performed, we construct SRTS, in the form of variations on a finite state machine model, according to random execution results to represent the sub-range transitions for internal variables. Since our goal is to achieve variable sub-range combination coverage, the total number of available sub-range combinations, i.e., the product of all C_i's ($1 \leq i \leq m$) could be a huge number when m gets bigger. Therefore, in order to avoid the state explosion problem, the SRTS is composed of a collection of separate sub-models $SRT_i (1 \leq i \leq m)$, each of which devotes to an individual internal variable V_i that has been selected for range partition using the algorithm described in Section 3.2. Each SRT_i can be represented visually as a directed-graph as shown in Figure 3.3(a), (b), which illustrates the *SRT* for internal variable V_i and V_j , respectively.

Each SRT_i can be formally represented as a 5-tuple structure: $SRT_i = \{S, \sum, T, s_0, F\}$, where :

- S is a finite non-empty set of states, given the internal variable V_i with C_i sub-ranges, S will correspondingly contain C_i states, each state represents one sub-range of V_i;

- $\sum$ is a finite set of input symbols. In this case, the input is set of vectors that represent the conditions related to the changes in the *input variables* such that the internal variable can transit among sub-ranges;

- T is a transition function T: $S \times \sum \rightarrow S$, which is shown as directed edges between states in Figure 3.3, indicating the data value of a given internal variable changes from one sub-range to another;

- S_0 is an initial state, i.e., the initial sub-range that V_i falls in;

- F is the set of final states, which refers to the goal sub-range of the internal variable V_i.

In other words, each SRT_i in SRTS models how the value of an internal variable varies according to the change of input variables.

More precisely, SRT_i defines the next state (i.e., sub-range) of variable V_i as a function of its current state (i.e., sub-range) and the input variable changing conditions. As shown in Figure 3.3, each transition directed from sub-range R_{ix} to sub-range R_{iy} ($1 \leq x, y \leq C_i$) is associated with a *direction vector* $<d_1, \ldots, d_H>^{[Rix,\ Riy]}$ and *transition weight* $w^{[Rix,\ Riy]}$, where H is the total number of the input variables in the program, and d_k ($1 \leq k \leq H$) is a *direction indicator* denoting the changing direction of the k_{th} input variable, e.g., d_k=0 means increase, d_k=1 means decrease, d_k=X means don't care. A don't-care means that any change in the k_{th} input variable will not influence the value of internal variable V_i based on the data dependence analysis. In addition, we append a transition weight $w^{[Rix,\ Riy]}$ with each direction vector to represent the level of confidence we have such that the value of V_i would be transitioned from sub-range R_{ix} to sub-range R_{iy} if the input variables that makes V_i falls into sub-range R_{ix} changes in the manner that compatible with the direction vector $<d_1,\ldots, d_H>^{[Rix,\ Riy]}$.

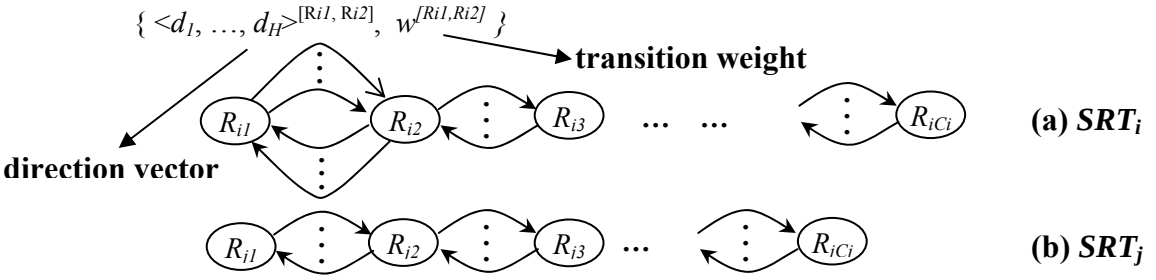


Figure 3.3 Sub-Range Transition System SRTs for Internal Variables V_i and V_j

The SRTs are constructed based on the execution results. Range partition determines the number of states in each SRT. And due to our partitioning principle, sub-ranges for any given internal variable are mutually exclusive, to simplify the model, only transitions between adjacent sub-ranges will be kept in each SRT. As to the input symbols, for each of the N execution runs, we recorded the values of both input variables and internal variables. After the range partition, N input vectors will be divided into C_i subsets, each of which is a collection of input vectors that make the value of V_i falls into one of its sub-

range. Through comparing the changing direction of each pair of input vectors with one from the subset associated with sub-range R_{ix} and one from sub-range R_{iy}, we can obtain all the possible direction vectors that can lead to the transition from R_{ix} to R_{iy} in the execution. The corresponding transition weight is calculated as: the ratio between the total number of simulated transitions from R_{ix} to R_{iy} that are caused by this direction vector and the total number of input vectors that transit V_i from R_{ix} to R_{iy}. We use the transition weights as guidance on how to apply the appropriate direction changing on the input variables in order to reach the goal sub-range.

3.3.2 Goal-Oriented Search Algorithm

Based on the SRTS model, we exploit a goal-oriented search algorithm to automatically generate test data so that the internal variable values would achieve high sub-range value combination coverage under the given range partition.

A sub-range combination is represented as a *m*-tuple $<r_1, \ldots, r_m>$, $(1<=r_i<=C_i)$, where each item within the tuple denotes a valid sub-range index for each internal variable. We also define the *sub-range distance* between two sub-range combinations as the summation of the absolute difference between corresponding sub-range index for each internal variable. Suppose the target sub-range combination is $<r_1^g, \ldots, r_m^g>$, the basic searching algorithm is as follows: first select an input vector from the existing test set that will make the internal variables fall into a sub-range combination: $<r_1^0, \ldots, r_m^0>$, which is closest to the target sub-range combination, or randomly generate one when the test set is empty. The word "closest" refers to the sub-range combination with the smallest sub-range distance to the target sub-range combination. Then, if $<r_1^0, \ldots, r_m^0> \neq <r_1^g, \ldots, r_m^g>$, within each SRT_i, choose a transition edge that points from sub-range r_i^0 to the direction of r_i^g. For example, if currently the internal variable is in the sub-range R_{i1}, and the target is R_{i4}, then the edges point from R_{i1} to R_{i2} would become candidates for selection. The transition edges are selected based on the weight of each direction vector, that is, within a total number of edge selections between the same two sub-ranges, the direction vector with higher weight will be selected with more chances proportional to its weight. Once a direction vector was chosen from each one of the SRT_i, the next step is to perform

conflict check. Since in order to cover a sub-range combination, all the internal variables involved have to be tuned simultaneously through changing the input variables in the way that is compatible to all the selected direction vectors. If there exists a conflict for the same input variable between two or more of such direction indicators, e.g., one chooses to increase input variable k while the other will decrease, then the above process is repeated to select new direction vectors from each SRT_i until a set of conflict-free direction vectors was selected. Next, all the direction indicators for the same variable are analyzed using a dominator rule (i.e., d_x=X is dominated by d_x=1 or 0) to determine the final changing tendency for each input variable. Finally, the input variables are tuned accordingly by random changing steps in the directions compatible with the final direction vector. Unlike the conventional goal-oriented search algorithm [38] which only one variable is changed at a time, we explore all the input variables at the same time to improve the efficiency of searching. The above process is repeated until either the target sub-range has been covered or the number of retries has exceeded the maximum value. We perform the above goal-oriented searching process by treating every sub-range combination as a separate target, until all the sub-range combinations have either been covered or failed.

3.4 Experimental Results

We performed experiments to investigate the effectiveness of the SRTS model in sub-range combination coverage-directed automated test generation as well as the capability of IVP testing in program fault detection.

3.4.1 Experiment Setup

The test benchmarks we used are a class of programs that are extended from the classical triangle program. As shown in Figure 3.4, function *ExtTrianlge* takes four integers x, y, z and w as inputs and output the triangle type. It is composed of two sub-modules: extended computation and conventional triangle module. The extended computation is added on top of the triangle, that performs computation on the input variables x, y, z and w and generates three integer internal variables A, B and C which are fed into the triangle module for triangle classification. We generate a class of 6 programs, T1 thru T6, with

different implementations in module "extended computation". In benchmarks T1-T3, *A*, *B* and *C* are just sequential functions of *x*, *y*, *z* and *w*, mostly are arithmetic computations, while the T4-T6 include conditional control operations. The average size of the benchmarks is 80 statements.

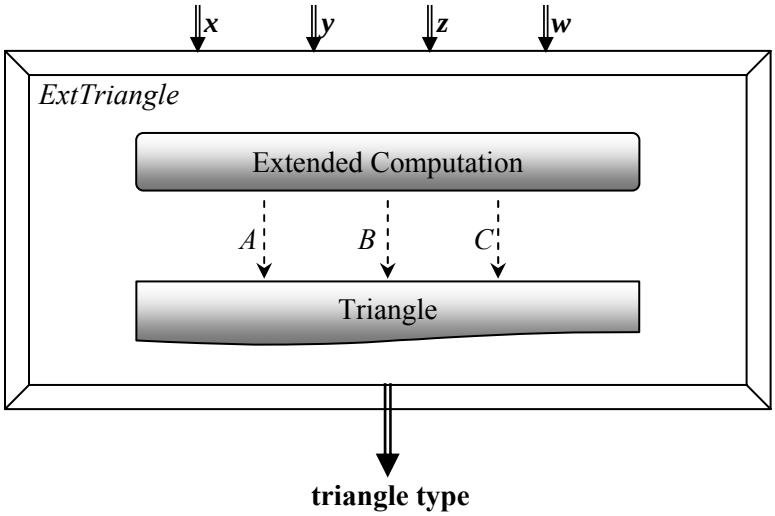


Figure 3.4 Benchmark Program Structures

We implanted each benchmark program into our implementation of dynamic-program-execution-based sub-range partition together with SRTS models for automated test data generation separately, and compared results among themselves as well as against the random test generation approach.

3.4.2 Sub-Range Combination Coverage

The experiments followed the process shown in the framework of IVP testing (Figure 3.2). First, internal variables *A*, *B* and *C* are selected as critical internal variables, and then we performed 1000 random executions for each benchmark program, followed by range partition and range merging processes. Finally, the range partition results are sent to SRTS model for test generation.

Table 3.1 gives the results of sub-range combination coverage for both SRTS generated test set and random test set. For fair comparison, both sets have the same size. According to the range partitioning algorithm described in Section 3.2, the final sub-range partition

results can be customized according to different parameter configurations. In our experiments, we tested with different values of range merging threshold *CNT_TH* (=50, 100, 120) and took an average on the coverage results. As we can see, the SRTS model-based test set can achieve up to 44% more average sub-range combination coverage than random test set which proves the effectiveness of our SRTS model in IVR coverage.

Table 3.1 IVR Coverage Results

Benchmarks	***Sub-Range Combination Coverage***	
	Random	***SRTS***
T1	57.3%	*79.0%*
T2	60.0%	*79.6%*
T3	45.3%	*89.0%*
T4	61.0%	*79.5%*
T5	35.5%	*50.0%*
T6	*58.1%*	*66.2%*

3.4.3 Mutant Coverage

As mentioned earlier, an important criterion to measure the effectiveness of coverage metric is its capability on fault detection. We apply the mutation testing approach to evaluate the quality of our IVR coverage-oriented test set. Table 3.2 lists the set of mutation operators we used in the experiments, which is a subset of the Mothra mutation operators [107].

Table 3.2 Mutation Operators

Operator	**Description**
CRP	Constant replacement
AOR	Arithmetic operator replacement
SVR	Scalar variable replacement
ROR	Relational operator replacement
LCR	Logical connector replacement

Then the test set is used to execute these faulty programs to check whether the mutant can be killed.

3.4.4 Fault Detection Capability

Both the random test set and SRTS generated test set were fed into the mutant programs of each benchmark to investigate and compare their effect on mutant killing.

Table 3.3 Mutants Killing Results

Benchmarks	**Total Mutants**	***Mutants Killed***	
		Random	***IVR-based***
T1	324	244	*262*
T2	337	270	*272*
T3	360	259	*290*
T4	375	300	*303*
T5	408	325	*330*
T6	*475*	*381*	*385*

The results are shown in Table 3.3. By applying the mutant operators listed in Table 3.2, an average of 80 statements benchmark can generate 300 to 400 mutants. As to the mutant killing, the results showed that IVRP-based test set outperformed the random test set. Note that in these programs, killing of a few additional mutants is noteworthy since they are hard corner cases. Moreover, benchmarks T1 through T3, which only contain sequential arithmetic computations in the extended module, demonstrate an average of higher degree of mutant killing capability over random test set than benchmarks T4 through T6, which also include conditional control statements. Thus, the IVP testing and IVR coverage metric are particularly suitable to be applied on the testing of programs involved with more arithmetic computations.

3.5 Conclusions

In this chapter, we have presented a new domain partition testing on critical internal variables in a program. A dynamic-program-execution-based testing framework is proposed for range analysis and definition based on the variable data value distribution as well as the program structure information related to the internal variables, which in turn introduced a new IVR coverage metric for sub-range combinations. Directed by the coverage metric, we proposed SRTS model to automatically generate test set. The experimental results on a class of extended triangle programs showed that the test set generated from SRTS model can achieve up to 44% higher internal variable range coverage than random test set, and mutation testing results showed that it also demonstrates better mutant killing capability than random test set, especially for programs that are arithmetic computation intensive.

Chapter 4

Mining-Based Invariant Extraction for Bounded Model Checking

In this chapter, we propose a novel framework that uses data mining in the execution data of the original program to extract a set of high-level potential property invariants according to the dynamic execution data of the software. These extracted invariants are in the form of single linear relations as well as complex implications among program variables. When these learned invariants are added to the bounded model checking instances of the software, they help to significantly reduce the search space. Experimental results showed that our approach can lead to up to an order of magnitude of speedup in software bounded model checking.

4.1 Motivation

As introduced in Section 2.2.2.2, Bounded Model Checking (BMC) translates the design under verification that violates certain specification(s) for a pre-defined finite depth into a propositional formula and thus convert the problem into a propositional satisfiability problem(SAT). However, the SAT-based BMC has limitations in the context of software program verification. One such limitation is during the process of conversion from a given program to one large propositional formula, in particular, when each variable is represented with a bit vector whose size is determined by the variable type, some high-

level information such as variable correlations and program structure features that can be easily gained in high-level programming languages are lost. Subsequently, extra burden is placed on the SAT-solver to reason about such relations at the Boolean level. For example, in the following sample code segment, suppose the loop index i is not changed within the loop body, it is quite obvious that property $x \geq y$ always holds at line L_1. However, it may not be evident when each variable has been converted into bit-vectors.

```
        for ( i = 0;   i < N;   i++ ) {
L0:       …
L1:       x = y + i;
L2:       …
        }
```

Figure 4.1 Example Code Segment 1 (Loop)

In addition, since the resulting propositional formula is simply a static translation of the original program semantics, no run-time implications on the variable correlations and path feasibilities are available. For example, in code segment Figure 4.2, suppose the operations between L_0 and L_1 on pointers p and q always get (*p) > (*q) at L_1, then, whenever L_1:(y<0) is evaluated to true, x is always greater than 0 at L_2 due to the relations between (*p) and (*q), consequently, L_5:(z>0) is always true, so path L_1-L_2-L_4-L_5-L_7 would be infeasible. Intuitively, adding these implied properties to the propositional formula may help further reduce the search space of model checking, thus improving the performance of SAT solving.

```
L0:  int  x,  y,  z,  *p,  *q;
     …                        //operations on pointers p and q
L1:   if (y < 0)
L2:      x = (*p) – (*q);
      else
L3:      x = (*p) + (*q);
L4:  z = x - y;
L5:  if (z > 0)
L6:       z = z + x;
          else
L7:       z = z + y;
```

Figure 4.2 Example Code Segment 2 (Control)

The intuitions came from the simple examples above have also been suggested by studies [14]. They showed that most of the properties established *during* verification are either invariants or depend crucially on invariants. Therefore, how easy these invariants, even trivial ones, can be deduced becomes a key issue to the effectiveness of the automated formal verification.

Among the two major categories of methods for learning invariants in a given program: through data flow/static analysis[4, 13] and through dynamic program execution[16, 17], our method falls into the latter category and uses data mining to detect invariants based on the execution profiles. Our mining-based framework exhibits more flexibility on the format of properties that can be learned compared to the conventional static program analysis approaches. Using mining, the learning scopes can span over relations among any kind of variables (including complex pointers) as long as they can be valuated, with relatively equal efforts. On the contrary, the static analysis is relatively conservative, and different dataflow equations or models have to be built for different kind of properties, moreover, it may vary significantly in the complexity of building and solving different models. In addition, our mining approach also overcome the limitation in existing dynamic invariant detection tools [16, 17] that only the relations among global variables and only at the entering and exiting locations of functions will be learned. Instead, we allow invariant detection at a finer granularity: at any program location and among all possible variables (both local and global). Hence, it increases the merit of these learned invariants to be utilized in the subsequent constraint reduction abstraction with respect to the property checking.

The remainder of the chapter is organized as follows. The architecture of our mining-based software verification framework is described in Section 4.2, followed by the discussion of detailed property mining and optimization algorithm in Section 4.3. Section 4.4 presented the experimental results. Finally, Section 4.5 concludes our findings.

4.2 Mining-Based Software Model Checking Framework

The key idea behind our mining-based software model checking framework is to discover and add appropriate property invariants (single as well as complex relations) to the

original program, so that the performance of model checking can be improved when verifying an assertion. The invariant extraction and model checking are somewhat independent. Figure 4.3 shows the overall architecture of the framework.

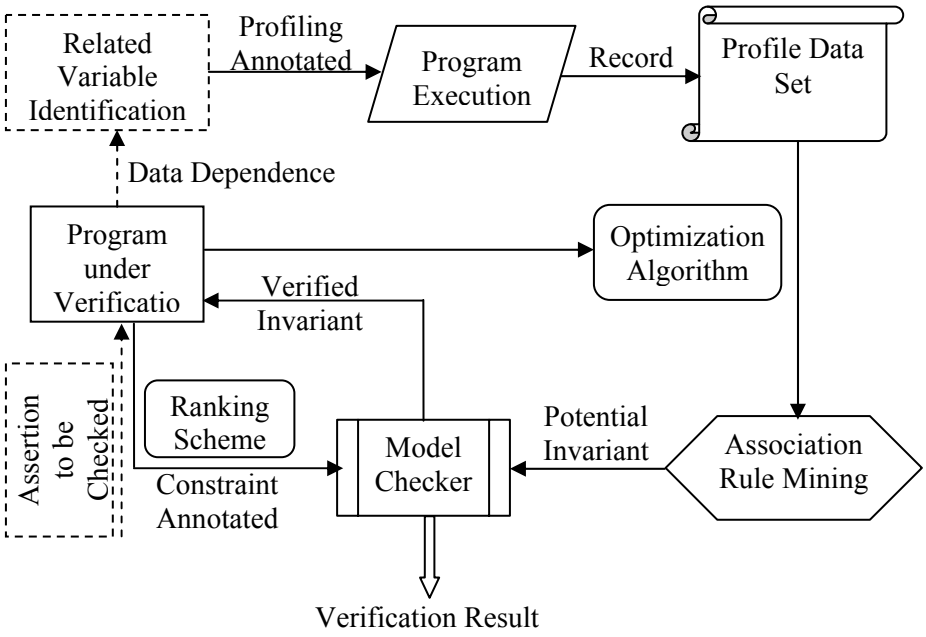


Figure 4.3 Mining-Based Software Model Checking Framework

As shown in Figure 4.3, given a software program P under verification and the target assertion R to be verified, we first proceed to the "Related Variable Identification" module to extract the related variables and their corresponding locations with respect to R, which allows us to focus on mining invariants among variables that are essential to the verification of R. Given an assertion R that consists of variables $X=\{x_1, \ldots, x_m\}$, for each $x_i (1 \leq i \leq m)$, we build the Use-Definition (UD) chains that link the use, U, of x_i to all its definitions, D. This chaining process is repeated for all the variables used at the definitions involved in D, until variable type definitions are reached. Next, by tracing backward along the UD chains starting from the assertion location for each x_i, all the variables that contribute to the value of x_i either directly or indirectly are said to be related to variable x_i. Take the code segment in Figure 4.2 as an example, if the assertion to be verified is '$z \neq 0$' at the program location right before line L_5, then the definitions for the use of z here would be chained to line L_4, this process is repeated for variables x and y used at L_4, until it reaches L_0, so the related variable set of z is $\{x, y, (*p), (*q)\}$.

If no target property is given or it is unknown, the related variable identification step will be omitted, as shown by the dashed-line enclosure in Figure 4.3. In such a case, all user-defined variables of interest are treated as related variables.

After identifying the related variables, the original program P is instrumented by adding profiling statements at the locations of the definitions and uses of each related variable. Let the annotated program be denoted as P_p. Each profiling statement records the values of all the related variables as well as the line number where it lies. Using the same example above, each profiling statement: **record_value(*lineIndex*, *x*, *y*, (**p*), (**q*))** will be inserted at line L_2, L_3, L_4 and before L_1 of the program in Figure 4.2. For those variables whose values are not available, it will record a "NA" in order to avoid exceptions during program execution. For example, for variable '$a[i]$' the profiling function checks the value of i before output $a[i]$ to ensure i lies in the bounds of array a. Next, program P_p is executed with randomly fed input values for a given number of times and the profiled values for related variables during dynamic execution are stored into a data set.

The resulting profile data set forms the database repository and will be sent to the association rule mining module for potential invariant mining of two pre-defined formats: single linear property and complex property implication. As introduced in Section 2.4, support and confidence are two parameters used to control the strength of association rules mined; these two values can be customized to tune the number and the quality of the properties that we discover. In addition, instead of mining a pure transaction data e.g., market-basket data, where each item is considered to be independent with each other, we are mining data set profiled from dynamic program execution. Therefore, an optimization mechanism is also used to help filter out redundant or incorrect property learned based on the correlations among variables and program control flow structure. The detailed property mining and optimization algorithms will be described in Section 4.3.

The potential property invariants discovered by association rule mining and optimization will then be sent to a model checker to check their correctness. In this process, each potential property invariant *Inv* is inserted as the only *assertion* to be checked to the

original program *P* at its corresponding location. Next, we verify whether the *Inv* holds by applying bounded model checking under an unwinding depth no less than the depth set for checking the original target assertion *R*. The *Inv* is considered as a true invariant and can be used as a constraint in subsequent steps if no counterexample can be found within the unwinding depth.

The next question is how to add these real property invariants as constraints to the program. Although additional invariants may help save the time of re-discovering these properties at the Boolean-level for the SAT-solver, the added constraints also may introduce extra variables and clauses that cause the size of the resulting propositional formula to grow bigger, especially when the invariant inserted is within a loop, in which case the invariant would appear repeatedly within each unrolled loop iteration. It is possible that the potential time savings brought by adding the learned invariants would be negated by the overhead of solving a much larger formula. Therefore, tradeoffs must be made between the number of constraints added and the formula size. Obviously it is impractical to identify the exact importance of an invariant in the verification of a given assertion, in this work, we apply a simple heuristic ranking scheme (addressed later in Section 4.3) that assigns a weight to each verified property invariant learned according to several factors, like the position of the assertion *R*, variables involved in the invariant, etc., such that, to some extent, the weight can measure the merit of each invariant with respect to the target assertion.

After ranking of the mined invariants, the original program *P* is then annotated by adding the verified property invariants whose weights exceed certain threshold as constraints, together with the target assertion *R* to be checked. The resulting formula is sent to the model checker for verification.

From the above description we can see that the mining-based software model checking framework not only provides mechanisms to closely integrate the invariant extraction process with the model checking with respect to certain assertion under verification, but also offers adequate flexibility for the user to control either the format or the quantity and quality of the invariants learned.

4.3 Property Invariant Mining and Rule Reduction Algorithms

4.3.1 Property Invariant Mining

In property invariant mining, we consider the linear relations among related variables with the same data type as well as between variable and some related constants. Given a target assertion R, suppose for the related variable set $V_R = \{v_1, \ldots, v_n\}$, $C_R=\{c_1,\ldots, c_m\}$ represents the set of related constants that have appeared in the program. We divide $V_R \cup C_R$ into subsets according to the data type of each variable/constant, like int, float, etc. Then those subsets whose size is 1 are removed, resulting in related set Ω_R, which consists of d subsets, $\Omega_R = \{\{e_{11}, \ldots,e_{1n1}\}, \ldots,\{e_{d1}, \ldots, e_{dnd}\}\}$, where the ith subset Ω_{Ri} contains n_i elements ($1 \le i \le d$).

After profiling the program with random inputs, a two-dimensional mining database D is constructed. Unlike regular transactions, the specific feature of the program execution data requires us to redefine the format and meaning of items and transactions in the mining database D. In D, each item is a 2-tuple in the form of:

$$I = <\wp::\lambda, x \blacklozenge y>, \blacklozenge \in \{>, \geq, <, \leq, =, \neq\} \qquad (4.1)$$

It is composed of two types of sub-items: 1) *location item* in the form of $<\wp::\lambda>$ where λ is a valid line number in the code, $\wp::\lambda$ means the program flow goes through line λ; 2) *relation item* in the form of $\{x \blacklozenge y\}$, where $\blacklozenge$ is one of the six comparison operators, and x, y are elements belonging to the same subset in Ω_R, and at least one of them should be a variable. All the possible combinations of the two sub-items constitute the set of items for the mining database. Thus, for the subset Ω_{Ri} containing n_i elements, there would be $6 * C_2^{n_i}$ (n_i choose 2) items at each profiled line. Suppose K lines in the program are annotated for profiling, the mining database would consist of $K*6*(C_2^{n_1} + \ldots + C_2^{n_d})$ items. Take the code segment in Figure 4.1 as an example, suppose line L_0, L_1, L_2 are involved for profiling, the related set $\Omega_R=\{\{x, y, i, 0, N\}\}$, then the item set for the database would be: $\{<\wp::L_k, x \blacklozenge y>, <\wp::L_k, x \blacklozenge i>, <\wp::L_k, x \blacklozenge 0>, <\wp::L_k, x \blacklozenge N>, <\wp::L_k, y \blacklozenge i>, <\wp::L_k, y \blacklozenge 0>, <\wp::L_k, y \blacklozenge N>, <\wp::L_k, i \blacklozenge 0>, <\wp::L_k, i \blacklozenge N>\}$, for k=1,2,3 and $\blacklozenge$ enumerates

each of the 6 comparison operators in (4.1). Each program execution sequence L_0-L_1-L_2 forms a transaction, and whenever a new iteration of the for-loop begins, it is treated as another transaction. In other words, a transaction is a sequential execution of the program control flow, with the line numbers involved being monotonically ascending and each line number occurs only once. Whenever a back jump happens during the program execution, a new transaction is started.

Table 4.1 gives an example mining database. The location item partitions all the data items (columns) into groups, such that all the relation items in the same group of $<\wp::\lambda>$ represent relations among *valid* variables at line λ. In this example, x, y, z and w are the valid variables at line 1 and 5. Due to the space, we omit some of the possible relation items at each line. Each row represents a transaction, which is composed of the corresponding value evaluation results (i.e., either *true*(1) or *false*(0)) for each data item according to the profile data collected during program execution.

Table 4.1 An Example Database for Property Invariant Mining

Trans.	**$<\wp::1>$**	**$\{x<y\}$**	**$\{x>w\}$**	**…**	**$<\wp::5>$**	**$\{y<z\}$**	**$\{z>x\}$**	**…**
1	1	1	1	…	1	1	1	…
2	1	1	0	…	1	1	1	…
3	1	1	0	…	1	1	0	…
4	0	0	0	…	0	0	0	…
5	1	1	1	…	1	1	1	…

In our work, we focus on mining two kinds of invariants: (a) single linear property (SLP) that holds at a certain line of the program (i.e., after the line is executed), and (b) complex linear property implication (CPI) that represents relations among variables at different lines of the program.

a) Single Linear Property Invariant Mining: A single linear property has the format of $(\wp::\lambda \Rightarrow x \blacklozenge y)$, which means $x \blacklozenge y$ always holds at line λ of the program. According to the item definition in (4.1), a single linear property depicts the relation between the two elements in an item tuple; hence, according to the A-Priori algorithm, the mining of such kind of property invariant becomes a frequent 1-itemset mining. The support and

confidence are computed as follows:

$$support(\wp::\lambda \Rightarrow x \blacklozenge y) = \text{Count}(\wp::\lambda \,\&\, x \blacklozenge y)/\text{Count}(\text{Trans})$$

$$confidence(\wp::\lambda \Rightarrow x \blacklozenge y) = \text{Count}(\wp::\lambda \,\&\, x \blacklozenge y)/\text{Count}(\wp::\lambda)$$

Therefore, all the items in the data set whose occurrence frequency for both location item $<\wp::\lambda>$ and relation item $(x \blacklozenge y)$ reach a certain support threshold s_{th} would be considered as SLP candidates. Table 4.2 listed the frequent 1-itemset computation result for s_{th}=0.5 for the example in Table 4.1.

Table 4.2 Frequent 1-ItemSet Computation for SLP Mining (s_{th}=0.5)

Candidate 1-itemsets	Count	Support	Freq. 1-itemset?
$<\wp::1>$	4	4/5=0.80	Y
{x<y}	4	4/5=0.80	Y
{x>w}	2	2/5=0.40	N
$<\wp::5>$	4	4/5=0.80	Y
{y<z}	3	3/5=0.60	Y
{z>x}	3	3/5=0.60	Y

For the case of SLP mining, we limit the condition part A of an association rule A⇒B to be a location item and the consequent part B a relation item in the group defined by A. Therefore, given the frequent 1-itemset computed in Table 4.2, we obtained three candidate association rules, as shown in Table 4.3. Given the confidence threshold c_{th}=0.7, all the three candidates are true association rules.

Table 4.3 Association Rule Derivation for SLP Mining

Candidate Assoc. Rules	Confidence	Assoc. Rule (c_{th}=0.7)?	SLP (c_{th}=1.0)?
$<\wp::1>\Rightarrow${x<y}	4/4=1.0	Y	Y
$<\wp::5>\Rightarrow${y<z}	4/4=1.0	Y	Y
$<\wp::5>\Rightarrow${z>x}	3/4=0.75	Y	N

However, since our goal is to first mine potential invariants before validating them, the

confidence threshold c_{th} for association rule generation should set to be 1.0, which guarantees the mined property always holds over the profile data set. Therefore, the association rule $<\wp::5>\Rightarrow\{z>x\}$ in the above example is not included as a SLP since there exist some cases where $\{z>x\}$ does not hold at $<\wp::5>$ (i.e., transaction 3 in Table 4.1).

In addition, due to the existence of conditional statements, loop statements, etc., in software, the execution rate of each different line in a program may vary significantly. For example, given the code segment below, the execution flow may reach line L_1 only once, while line L_2 could be exercised N times. As a result, the occurrence frequency of different items involving different lines in the mining database would differ accordingly based on the definition of transaction. Therefore, in (4.2), the number of transactions which occurs at $\wp::L_1$ could be much smaller than that at $\wp::L_2$ since each loop iteration starts a new transaction.

```
L1:     x = y + 1;
        do{
L2:         y = x + i;
            …
        } while (i < N)
```
(4.2)

So, we usually set the support threshold s_{th} for single linear property mining to a relatively low value to avoid overlooking the outside-loop properties, like properties at L1 in (4.2).

b) Complex Property Implication Mining: A complex property implication is defined as:

$$\{<\wp::\lambda_1> \,\&\, (x_1 \blacklozenge y_1)\} \Rightarrow \{<\wp::\lambda_2> \,\&\, (x_2 \,\square\, y_2)\} \tag{4.3}$$

where $\lambda_1 < \lambda_2$ and $\blacklozenge$, $\square \in \{>, \geq, <, \leq, =, \neq\}$, $\{x_1, y_1\}$ and $\{x_2, y_2\}$, respectively, are elements from the same subset in Ω_R, the two subsets may not necessarily be the same. This kind of property implication inspects implicit knowledge among variables spanning over different program locations, that is, if $x_1 \blacklozenge y_1$ is true at line λ_1, $x_2 \,\square\, y_2$ is definitely true at line λ_2, which makes it harder but valuable to discover.

According to the definition in (4.3), CPI mining implies a frequent 2-itemset computation. Take the Table 4.1 again as an example, by enumerating all pairs of *valid* frequent 1-items in Table 4.2, it forms the candidate 2-itemsets. In this case, a valid 1-item is in the form of (I_l &I_r), where I_l is a location item and I_r is a relation item in the group defined by I_l, for example, (<℘::1>&{x<y}) is a valid frequent 1-item. We can then compute the frequent 2-itemset based on support values of each candidate 2-itemsets. The results are shown in Table 4.4 for s_{th}=0.5.

Table 4.4 Frequent 2-itemset Computation for CPI Mining (s_{th}=0.5)

Candidate 2-itemsets	Count	Support	Freq. 2-itemset?
[<℘::1>&{x<y},<℘::5>&{y<z}]	4	4/5=0.80	Y
[<℘::1>&{x<y},<℘::5>&{z>x}]	3	3/5=0.60	Y

From the frequent 2-itemsets, we can generate the candidate association rules in the form of $A \Rightarrow B$. For the case of CPI mining, we limit A to contain smaller location item than B. For example, <℘::1>&{x<y}⇒<℘::5>&{y<z} is a candidate association rule. As shown in Table 4.5, given the confidence threshold c_{th}=0.7, both candidates are returned as association rules. As in the single linear property invariant mining, CPIs should also be generated using confidence threshold c_{th} =1.0. Therefore, in this example, only <℘::1>&{x<y}⇒<℘::5>&{y<z} meets the requirement of CPI.

Table 4.5 Association Rule Derivation for CPI Mining

Candidate Assoc. Rules	Confidence	Assoc. Rule (c_{th}=0.7)?	CPI (c_{th}=1.0)?
<℘::1>&{x<y}⇒<℘::5>&{y<z}	4/4=1.0	Y	Y
<℘::1>&{x<y}⇒<℘::5>&{z>x}	3/4=0.75	Y	N

Unlike in single invariant mining, the support threshold s_{th} for complex property implication mining usually is set to a relatively higher value to avoid mining an implication from an outside-loop property to an inside-loop property which in most cases is proven false or useless. Take the same example in (4.2), property implication {<℘::L_1>& (x♦y)} ⇒ {<℘::L_2> & (x¤y)} should be avoided, and setting a high s_{th}

value would probably filter out such implication due to the relatively high support difference between the condition and consequent property.

4.3.2 Mining Rule Reduction Algorithms

Since data mining only outputs potential property invariants that hold over the execution data set, some of them may not be true under a different execution trace or may be redundant. Hence, it is necessary to filter out those less valuable ones to reduce the burden of verifying them. We applied several rule reduction algorithms on the mining results by utilizing the correlations among items and transactions as well as the program structure.

1) Mining Rule Reduction Algorithms for Single Linear Property Invariants:

Rule 1.1: Redundant property removal:

Remove those single invariants that can be implied by other learned single invariants based on the semantic relations among comparison operators. For example, if both P_1: $\{\wp::\lambda \Rightarrow x<y\}$ and P_2: $\{\wp::\lambda \Rightarrow x\leq y\}$ are mined at line λ, P_2 will be removed because $x<y \Rightarrow x\leq y$. Similarly, we can remove both $x\leq y$ and $x\geq y$ if $x=y$ is true at the same line, etc.

Rule 1.2: Equivalent property removal:

If a linear relation R_1 holds at both line i and line j, where $i<j$, and both variables involved in R_1 are not re-defined within the program scope ($i<L\leq j$), then R_1 at line i and R_1 at line j are considered as equivalent properties. Rule 1.2 will remove all the equivalent properties except the one at the minimum line number. As the example shown in (4.4), $\{<\wp::L_1>\Rightarrow (a<b)\}$ and $\{<\wp::L_n>\Rightarrow(a<b)\}$ are equivalent properties, and $\{<\wp::L_n>\Rightarrow (a<b)\}$ will be removed according to Rule 1.2.

```
L1:     a = b-N;                    //a<b holds
        …          } a, b not
        …          } changed                          (4.4)
Ln:     c = a+b;                    //a<b holds
```

2) Mining Rule Reduction Algorithms for Complex Property Implications:

Rule 2.1: Potential fact implication removal:

Given a property implication P_1:{<$\wp$::λ_1> & ($x_1 \blacklozenge y_1$)} $\Rightarrow$ {<$\wp$::λ_2> & ($x_2 \square y_2$)}, if the consequent part {<$\wp$::λ_2>$\Rightarrow$ ($x_2 \square y_2$)} is mined as a potential single linear property invariant, P_1 can be removed because implying a fact would not be interesting.

Rule 2.1 is a conservative heuristic, since at the mining phase, we only have confidence on the correctness of {<$\wp$::λ_2>$\Rightarrow$ ($x_2 \square y_2$)} to some extent, but can't guarantee it. However, it can be used as an effective rule when too many outputs are returned by the mining algorithm.

Rule 2.2: Equivalent implication removal:

Similar to Rule 1.2, this rule removes the property implication P_1:{ <$\wp$::λ_1> & ($x \blacklozenge y$)} $\Rightarrow$ {<$\wp$::λ_2> & ($x \blacklozenge y$)} if both properties involve the same relation between the same variables whose values are not changed between line λ_1 and λ_2, since the condition and consequent part of P_1 are semantically equivalent. For example: {<$\wp$::L_2> & ($x>y$)} $\Rightarrow$ {<$\wp$::L_4> & ($x>y$)} will be removed in code segment in Figure 1.2 since x and y are not re-defined between L_2 and L_4.

Rule 2.3: Transitive implication removal:

When the same linear property is implied by different properties, i.e., P_1: {<$\wp$::λ_1>& ($x_1 \blacklozenge y_1$) $\Rightarrow$ <$\wp$::λ_3> & ($x_3 \square y_3$)} and P_2: {<$\wp$::λ_2> & ($x_2 \bullet y_2$) $\Rightarrow$ <$\wp$::λ_3> & ($x_3 \square y_3$)}, where $\lambda_1<\lambda_2<\lambda_3$. If P_3: {<$\wp$::λ_1>&($x_1 \blacklozenge y_1$) $\Rightarrow$ <$\wp$::λ_2> & ($x_2 \bullet y_2$)} is also mined, then remove P_1, since P_1 can be implied from P_2 and P_3, according to transitive feature of implication.

Although these are straightforward rules, in the context of association rule mining, where no relations are assumed to exist among items and among transactions, they work as complementary techniques to establish the dependencies within the software transaction data, so that the quality of the properties mined can be improved.

4.3.3 Ranking Scheme

As discussed in Section 4.2, in order to enhance the effects of invariants learned from mining in software model checking, we have to make tradeoffs on the number of constraints to be added to the program and the resulting propositional formula size.

Given P_c as the target assertion to be checked at line L_c, which consists of variables $\{z_1,\ldots,z_k\}$; a single property P_s: $\{<\wp::\lambda>\&\ (x>y)\}$ is a verified invariant learned from data mining, where $L_c>\lambda$. Intuitively, we assume the following three parameters are factors that influence the assertion verification: 1) the statement distance between P_s and P_c; 2) the variable distance between P_s and P_c; 3) the occurrence frequency of variables in P_s. Particularly, the closer the P_s is to the P_c, the more closely related the variables in P_s are to the variables in P_c and he more locations the variables in P_s are involved in the code before P_c, the more crucial is the P_s in assertion verification. Therefore, we define the weight of the P_s as:

$$weight(P_s) = c_1\times(H - (L_c - \lambda)) + c_2\times\Sigma_i(dist(x, z_i) + dist(y, z_i)) + c_3\times(freq(x) + freq(y)) \quad (4.5)$$

$$\text{where} \qquad dist(a, b) = \begin{cases} H, & \text{if } a=b \text{ or } b \text{ is const} \\ H - (\text{level distance between } a \text{ and } b \text{ in the UD chain of } b), & \text{if } a\neq b \end{cases}$$

and $freq(a)$ = number of lines that a appears between λ and L_c

c_1, c_2 and c_3 are weighting coefficients that satisfy $0 \le c_1, c_2, c_3 \le 1$ and $c_1+c_2+c_3=1.0$. H is a constant which is large enough to guarantee both the first part of the *weight* and *dist* function to be non-negative. The three addends from left to right in the expression (4.5) reflect the above three factors, respectively, associated with weighting coefficients. The statement distance is simply the line difference between the P_s and the P_c. The level distance between a and b in UD chain of b is calculated as the number of steps needed to reach a from b by tracing backward along the UD chain of b, if a directly or indirectly contributes to the value of b; otherwise, the level distance is set to be H. Take the code segment in Figure 1.2 as an example, the level distance between z at line 5 and x at line 2

or line 3 is 1. Finally, the occurrence frequency of a variable is evaluated by the number of lines that involve this variable, i.e., either used or defined, between line λ and L_c.

Similarly, for property implication P_i: $\{<\wp::\lambda_1> \& (x_1 \blacklozenge y_1)\} \Rightarrow \{<\wp::\lambda_2> \& (x_2 \square y_2)\}$, the weight is computed as:

$$weight(P_i) = c_4 \times weight(<\wp::\lambda_1> \& (x_1 \blacklozenge y_1)) + c_5 \times weight(<\wp::\lambda_2> \& (x_2 \square y_2)) \quad (4.6)$$

where c_4 and c_5 are weighting coefficients such that $0 \leq c_4, c_5 \leq 1$ and $c_4 + c_5 = 1.0$. In other words, our ranking scheme is an offline evaluation on invariant importance through weight computation, which plays an essential role in balancing the gains and overhead of invariant-based model checking.

4.4 Experimental Results

4.4.1 Experiment Setup

We use CBMC (C Bounded Model Checking) [6] as the model checker with the underlying SAT-solver using MiniSat [19]. Several C programs on data structure and algorithms are selected as benchmarks, including: bubble and shell sorting algorithms and binary and Knuth-Morris-Pratt (KMP) searching algorithms.

These benchmarks annotated with profiling statements are first sent to dynamic program executions either for N=100 times on randomly generated inputs or when transaction count exceeds 5000 in the mining database, whichever comes first, all the profile data are stored into a file. In these experiments, we assume the final assertion to be checked is unknown in advance, so that the related variable sets are determined by programmers.

4.4.2 Data Mining and Rule Reduction

We implemented the proposed association rule mining algorithms and partial rule reduction algorithms in C++, some of the mining rule reduction algorithms and ranking scheme are currently done manually due to the relatively small program sizes, but they can be easily automated. For single linear property mining the support threshold s_{th} is set as 0.3, for complex property implication mining, the support threshold s_{th}=0.8,

confidence threshold for both property mining is 1.0.

Table 4.6 Results for Mining and Rule Reduction Algorithms

Benchmarks	SLP		CPI		Mining Time (min)	Verified Invariants		
	Mine	Opt	Mine	Opt		SLP	CPI	(%)
BubbleSort	16	5	68	1	0.8	5	1	100
ShellSort	39	11	164	0	0.9	11	0	100
BinarySrch	28	11	66	4	0.1	11	4	100
KMPSrch	57	14	262	13	1.0	4	10	52

Table 4.6 shows the number of properties learned purely through association rule mining and after rule reduction algorithm is applied for both Single Linear Properties (SLP) and Complex Property Implications (CPI). As shown in the table, with our rule reduction algorithms, the number of properties learned by data mining can be reduced significantly in both SLP and CPI, some can even achieve more than 95% of reduction, like the complex property implications in *ShellSort* and *KMPSrch*. It further proves that without the knowledge of the dependencies among items and transactions, data mining would generate quite many redundant properties for software database, of which a significant part can be covered by the categories of our mining rule reduction algorithms. As given in Table 4.6, it takes quite small running time for database generation, mining and rule reduction, for example, only 50 seconds are needed for mining on a database with size of 5117(Trans)×66(Items) for *BubbleSort*.

The learned properties reported in Table 4.6 are just potential invariants; thus they cannot be used as constraints without being verified. We asserted the potential properties after rule reduction into the program and used CBMC for correctness verification. The last column (%) of Table 4.6 gives the resulting percentage of verified invariants; all the benchmarks reached 100% of accuracy rate except the *KMPSrch*, which demonstrates that the threshold settings in our association rule mining are effective in filtering out the false positives. Although the needs for potential properties verification introduce additional overheads to software model checking, the advantage is that, a one-pass mining and invariant verification process can satisfy the requirements for different

assertion checking, especially when critical variables are chosen as related variable set, no iterative learning process is required.

4.4.3 Model Checking

Before adding the proved invariants to the original program as constraints, we utilized the simple ranking scheme on the invariants in order to evaluate the quality with respect to the assertion to be checked. Take *BubbleSort* as an example, we compute the weight value for each of the six verified invariants using formula (4.5) and (4.6) where we selected c_1=0.2, c_2=0.3, c_3=0.5, c_4=0.4, c_5=0.6 and H=5. Then, we insert each of them as constraint separately to the program under verification and run model checking. The results are shown in Table 4.7.

Table 4.7 Property Rankings for BubbleSort

Property Invariants	Rank	Weight	Verification Time(sec) for N			
			15	20	25	30
P2(SLP)	1	6.6	5.2	19.3	193.7	353.3
P1(SLP)	2	6.1	1.3	5.5	163.7	1504.3
P5(SLP)	3	6.1	3.2	36.1	365.2	1300.9
P3(SLP)	4	5.2	2.8	54.7	312.2	1237.1
P6(CPI)	5	4.0	6.1	199	1077.2	3298
P4(SLP)	6	3.2	16	98.5	2121.9	4078.8

We can see that this simple ranking scheme does provide rough estimation on the value of the invariant in terms of assertion verification, although it may not be always accurate for individual cases. For example, P_2, P_1 and P_5 who can achieve the best results are ranked among the top positions; P_6 and P_4 which are ranked at the bottom did demonstrate much worse performance than others, especially when unwinding bound N increases.

The verified invariants are added to the original program with the target assertion to be checked using *__CPROVER_assume* provided by CBMC to restrict the search space and apply assume-guarantee reasoning. Considering the program size, we limit no more than

2 invariants will be added at the same time to the program in these experiments. P_1 and P_2 are among the top ranked invariants for each benchmark, P_1&P_2 means both P_1 and P_2 are assumed at the same time. Since all the benchmarks are loop-based algorithms, Table 4.8 also lists the results of model checking under different unwinding depth settings.

Table 4.8 Model Checking Results with Added Invariants

Benchmarks	Unwind depth	Verification Time (sec)				Sat?
		Orig.	P_1-only	P_2-only	P_1&P_2	
BubbleSort	20	59	**6**	**27**	**14**	Sat
	25	325	**164**	428	**153**	
	30	1245	1504	1392	1979	
ShellSort	20	158	**154**	**76**	**65**	Sat
	25	545	**452**	573	607	
	28	1473	**1314**	**1071**	2055	
BinarySrch	10	42	**37**	140	42	UnSat
	15	580	**540**	**134**	957	
	20	709	**386**	**373**	1305	
KMPSrch	10	16	**10**	23	18	UnSat
	15	154	**136**	**149**	**133**	
	20	805	**693**	**615**	**523**	

From the results we can see that, with the added verified mined invariant(s), in most cases (in bolded), the verification time of the real assertion (either Sat or UnSat) can be improved significantly, some can reach up to an order of magnitude speedup (e.g., P_1-only case for *BubbleSort* with unwinding depth at 20). These results not only demonstrate the effectiveness of our mining-based framework in learning and selecting valuable invariants that are crucial to the assertion verification, but also prove the importance of high-level invariant extraction for SAT-based software model checking. On the other hand, it can also be observed that adding multiple invariants at the same time do not always guarantee better performance than adding individual ones separately, especially when the unwinding bound gets larger, for example, in *BinarySrch* when unwinding

depth is 20, P_1&P_2 turns out to be much slower than P_1-only, P_2-only and the original cases. This is probably because more invariants lead to a much larger propositional formula size, such that the extra cost incurred on SAT-solver exceed the potential savings that may brought by the invariants. Therefore, it is important to balance between the added invariants and the formula complexity.

4.5 Conclusions

We have presented a novel framework for software verification that combines data mining mechanism with the bounded model checking. High-level single property invariants and complex property implications are extracted via an association rule mining technique according to the profile data of the dynamic execution of the software programs. Mining rule reduction algorithms help to further filter out those not-so-interesting properties to control the quality of the property learned, at the same time, to release the burden of property verification. The mining-based framework provides straightforward and flexible mechanism for knowledge discovery. Experimental results showed that by adding the extracted and verified properties as constraints to the program, it can help improve the performance of software bounded model checking significantly through knowledge and search space reduction.

Although demonstrated only for the learning of two kinds of properties: single linear property at specific program location and complex property implications that cross different locations in the program, the proposed framework has the flexibility to be extended to other format of properties, which are potential future directions.

Chapter 5

Ant Colony Optimization Directed Under-Approximation Based Program Abstraction

In this chapter, we explore an under-approximation based program abstraction technique with the aim to improve the performance of software bounded model checking by falsifying the target property more quickly. In particular, we utilize the population-based meta-heuristic Ant Colony Optimization (ACO) [57] to direct the construction of under-approximate abstract state space. ACO is widely applied to find approximate solutions for optimization and search problems through the cooperation and adaption of artificial ants that simulate the food hunting behaviors of real world ants. Unlike existing methods that use ACO for the sole purpose of test data generation [58, 59, 60, 61], in our approach, we leverage the power of ACO to facilitate the construction of path-level abstraction for software model checking. The basic idea is as follows: given a property to be checked, artificial ants are sent out to explore the concrete state spaces such that the program execution flows would be led towards traversing the promising state space with respect to the property falsification. The program traces gathered by the ants would then be used to construct program under-approximation by biasing the control-flow that ignores less

useful statements. Subsequently, we adopt an iterative abstraction-based software bounded model checking framework on the reduced program until either a counterexample is found or the property is verified. Experimental results demonstrated the effectiveness of the approach in achieving dramatic performance enhancement in software verification, especially for property falsification.

The remainder of the chapter is organized as follows. The motivation of our work is introduced in Section 5.1 with an example. Section 5.2 discussed the related work from various related areas. The ACO-directed program abstraction process is described in Section 5.3. Section 5.4 presents the new bounded model checking algorithm through iterative program abstraction. Experimental results are given in Section 5.5. Finally, Section 5.6 concludes the chapter.

5.1 Motivation

Intuitively, the complexity of solving a large and intractable problem usually can be mitigated through the integration of results from solving several smaller and simpler problems. In the context of software model checking, rather than exploring the original huge state space, it may be beneficial to partition the original (large) state space of a given program into smaller subspaces, i.e., building an under-approximation by imposing feasible assumptions. Then, we collectively integrate the verification results from each abstract subsystem. In particular, at the initial phase of code design, when falsification (bug finding) is the major task of model checking, it is highly possible that by intelligently identifying an under-approximate model, the corresponding reduced state space would be sufficient to falsify a target property. In this regard, we focus on property falsification as the major aim in the rest of the chapter.

Conventional program reduction includes static program slicing, where code portions that have no effect on the semantics of the target point of interest get deleted. However, program slicing usually results in an exact abstraction of the original program with respect to the target, and irrelevant statements are removed solely based on the program data and control flow structure. For example, in Figure 5.1, program slicing cannot

remove any of the statements shown before the assertion because all these statements within both the then- and else- branches directly or indirectly contribute to the values of x and y, which are the variables involved in the assertion. Apparently, the reduced program obtained via static program slicing can only provide modest simplification over the program and thus limits the potential gains it could bring to the software verification. Instead of relying only on the static program structure information, we seek for a more aggressive abstraction approach to obtain under-approximations under the guidance of concrete state space exploration, i.e., dynamic program execution. Although only a partial state space is visited during the execution, useful information regarding the characteristics of program behaviors, for instance, variable relations, branching directions, etc., can still be learned with carefully designed program inputs. In addition, unlike the conservative nature of static approaches that must be sound for every single input, dynamic execution is a good candidate for identifying an approximation of promising areas and further narrowing down the size of the resulting reduced program, so that it can be utilized by the model checking to achieve higher performance improvement.

```
    if  (0 < x ≤ N && y ≤ M)          //e.g., a cex occurs when x=N, y=0
        x = f1();
        y = f2();
    else                        // x<y holds after the execution of else-branch
        while ( cond )
            x = f3();
            y = f4();
        ……                                        // operations on x and y
lk: assert(x < y);
```

Figure 5.1 Example for Property Falsification

Using the code fragment of Figure 5.1 again, suppose $x<y$ at line l_k is the target safety property for which few counterexamples (e.g., $x=N$, $y=0$) can be found. Further, let us assume that the property can only be falsified via the then-branch of the if-then-else (ITE) code. Without abstraction, the entire program will be converted into one single Boolean formula to be solved in the context of software BMC [6]. Moreover, if the else-branch of the ITE contains a large bulk of code and complex program control structure, like the while-loop shown in Figure 5.1, the size of the resulting formula might grow quite large, which in turn could significantly degrade the performance of model checking.

On the contrary, if we are able to identify promising code areas from dynamic execution that provides hints to where counterexamples most likely reside (e.g., the then-branch) beforehand, it would be helpful to enhance the efficiency of model checking by taking advantage of the reduced state space exploration with those less useful areas removed.

5.2 Related Work

Our work touches upon several research areas, including: program abstraction, dynamic execution-guided software model checking, and the application of ACO as an optimization methodology for handling the problems in software testing and verification. We will discuss the corresponding background and previous work in these areas in the following subsections.

5.2.2 Program Abstraction Based Verification

A substantial amount of research has been conducted on the application of abstraction in software model checking, ranging from model level abstraction (e.g., partial order reduction [68]) to program code level abstraction (e.g., predicate abstraction [34, 69]). Our work, instead, shares similarities with abstraction techniques based on under-approximation refinement [62, 71] or exact program transformation [70, 71]. Bryant et al. in [62] presented an abstraction-based approach for deciding the satisfiability of finite-precision bit-vector arithmetic, where bit-vector variables were encoded with fewer Boolean variables than their width. Such an under-approximation is interleaved with over-approximation during the iterative refinement process until a solution is found. On the other hand, [70] presented various program slicing techniques as program transformation based exact abstraction for software model checking, including conditioned and amorphous slicing. The abstraction technique proposed in [71] integrated slicing with under-approximation and was applied to software bounded model checking. Our work explores a different abstraction scheme which is performed at the *path-level* of the program. A reduced program is obtained by pruning away the less promising branches from conditional statements with respect to the target property checking. On the other hand, with the help of ACO-directed program structure traversal, our approach intends to gain more aggressive slicing on the entire state space than the

traditional program slicing, therefore, higher performance savings could be expected with the significantly reduced search space.

5.2.3 Dynamic Execution Guided Software Model Checking

There has been an emerging trend that combines software testing/dynamic program execution with formal verification of software, in which the target model is either abstracted or in its original form. These dynamic techniques work as heuristics to guide state space traversal towards promising areas, so as to alleviate the state space explosion problem in model checking. For example, CMC [63] applied direct program executions to guide the bug-finding. Another category of research focuses on integrating the concrete execution-directed under-approximation with over-approximation based abstraction techniques [64, 65, 66, 67], where dynamic executions or testing are used to guide the reasoning or refinement decisions made in abstract models.

The above approaches require the concrete states to be mapped to abstract states, with the concrete states providing hints for the traversal though the abstract state space. While in our work, concrete execution results are used to direct the code-level abstraction through under-approximation. Furthermore, instead of using randomly generated program inputs, we feed the dynamic program execution with inputs selected by ACO so that execution flows can be effectively led down to the promising areas with respect to the property to be checked.

5.3 ACO-Directed State Space Exploration

In this section, we present in detail the proposed ACO-directed concrete state space exploration algorithm with the goal of revealing useful structural information in the code with respect to the target property. Such information will be utilized to guide the subsequent construction of under-approximation based program abstraction for property verification.

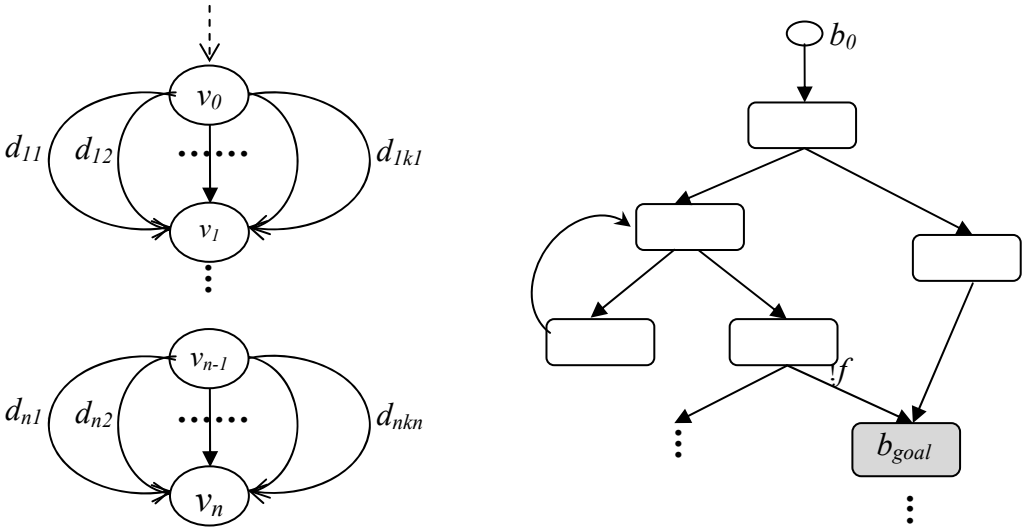
5.3.1 ACO-Directed State Space Exploration Model

Suppose the program under verification P consists of n input variables: $I=<x_1, \ldots, x_n>$, each of which lies within an input domain D_i:$[Di_{min}, Di_{max}]$, where $1\leq i\leq n$. Let the target property to be checked, A, be represented as $\{f\}@l$, where f is a Boolean formula defined over variables in P, and l is the line number in the code. In other words, property A checks whether f holds at line l. The ACO-directed state space exploration problem involves the process of finding an appropriate input vector: $T=<t_1, \ldots, t_n>$ for P, such that $t_i \in D_i$ $(1\leq i\leq n)$ and $\neg f$ is true at line l when the program is run with T as its inputs. In short, ACO tries to lead the program execution to falsify A. In this chapter, we assume the target property is a hard corner case such that it is not so easy to find such a T by directed program execution alone within a pre-defined number of iterations.

In essence, the first step of applying ACO is to model the problem with appropriate representations so that the artificial ants can navigate toward the target. As the ultimate goal of our ACO-directed state space exploration is to identify promising structures in P that constitute state spaces that can most likely falsify the property to be checked, we are focusing more on the paths/traces that the artificial ants have explored during the solution construction rather than focusing on the program input values. Hence, we divide the problem representation into two graphical sub-models: *Input Space Model* (ISM) and *Control Flow Model* (CFM).

For the input space, we apply a similar model as in [61]. The ISM of P is represented as a directed graph $G_{in}(V, E)$ consisting of $n+1$ vertices $<v_0, \ldots, v_n>$ (Figure 5.2(a)), where v_0 is the starting node, and any other vertex v_i $(1\leq i\leq n)$ in the graph corresponds to an input variable x_i in I. In order to model the value selection activity for input variables during test generation, for every input variable x_i in I, we further partition its input domain D_i evenly into k_i sub-domains:$<d_{i1},\ldots,d_{iki}>$, where $D_{i,min}\in d_{i1}$, and $D_{i,max}\in d_{iki}$. Correspondingly, every vertex v_{i-1} $(1\leq i\leq n)$ in G_{in} is connected with its next vertex v_i through k_i edges, representing k_i possible paths that can be selected from and traveled along by ant agents. If the j^{th} $(1\leq j\leq k_i)$ edge is taken by an ant from v_{i-1} to v_i, it implies that the current value of input variable x_i is ranged within its j^{th} sub-domain: d_{ij}. Therefore,

a valid input vector of P can be determined when a directed path from v_0 to v_n is traversed by an ant, with each node to be visited only once and only one edge chosen between every pair of adjacent nodes.



(a) Input Space Model (b) Control Flow Model

Figure 5.2 ACO-Directed State Space Exploration Model

While the ISM is used as the real site for ants to navigate through, the CFM works as a *virtual* space that captures the paths exercised by the ants virtually when the program is executed with a valid input vector consistent with a real path in the ISM. As shown in Figure 5.2(b), the CFM $G_{cfg}(B, C)$ is built directly on the control flow graph (CFG) of P. Each node b_i represents a basic block in P and the edge $c_{i,j}$ may be labeled with conditional guards that control the transition from node b_i to b_j. A virtual path in CFM starts from the entry node b_0 and ends at a leaf node which has no descendants. Given a target property $\{f\}@l$, we first instrument P with an ITE statement inserted at line l: *if* (!f) *then* {b_{goal}}; *else* {*original code blocks in P when f is not violated*}. The task of ants is to lead the program execution towards the target block b_{goal} in CFM during the input selection process when they walk around ISM.

The basic idea of the ACO-directed state space exploration algorithm is as follows: At every iteration, an ant is sent to the ISM and performs the following two tasks: 1)

Construct promising program structure by traversing the state space with an appropriate input vector, which is obtained using a probabilistic transition rule based on the amount of synthetic pheromone along the trails in both real and virtual models in Figure 5.2; 2) Update pheromone intensities on both trails based on a fitness evaluation; 3) Apply daemon actions when necessary to adjust pheromones and guide the search process offline from a global perspective. The above process is repeated until the target block is reached or a pre-defined number of iterations are exceeded.

5.3.2 Stochastic Promising Structure Construction

In order to construct promising program structures with respect to the target b_{goal}, a new ant enters ISM and tries to select an appropriate input vector (i.e., a path in ISM) that would most likely trigger the program execution flow down to b_{goal}. As the pheromone levels indicate the quality of a route, the path selection of the new ant is guided by the pheromone trails left by preceding ants. We associate a 3-tuple pheromone vector: <*phrm*, *desire*, *prob*> with every edge *e* in the ACO model (ISM and CFM), where the *phrm* represents the pheromone intensity accumulated on the edge, *desire* denotes the desirability (or quality) of the edge in terms of target reaching and the *prob* indicates the probability for the incoming ants to follow *e*. Suppose a vertex v_i has l_i outgoing edges, the probability $prob_{ij}$ that an ant will take the j^{th} ($1 \le j \le l_i$) edge out of l_i possible paths is calculated as in Formula (5.1):

$$prob_{ij} = \frac{\left(phrm_{ij}\right)^{\alpha}\left(desire_{ij}\right)^{\beta}}{\sum_{k=1}^{l_i}\left(phrm_{ik}\right)^{\alpha}\left(desire_{ik}\right)^{\beta}} \qquad (5.1)$$

where α and β are parameters used to control the influence of $phrm_{ij}$ and $desire_{ij}$, respectively.

The pheromone vectors define a stochastic transition rule for edge selection between two adjacent nodes in ISM and branch taking in CFM, such that edges with higher integrated effect of desirability and pheromone levels (i.e., quality) are more likely to be followed. According to the relationship between the edges in ISM and the sub-domains of input

variables, we can thus create a valid input vector after the ant chooses a path in ISM from v_0 to v_n by assuming that all the possible input values within a sub-range defined along each edge is uniformly distributed.

We use the Monte-Carlo method to simulate the ants' path selection activity. It randomly samples inputs based on the probabilistic pheromone distribution over the trails in ISM, until the corresponding virtual path obtained (after applying the sampled inputs to the program) also satisfies the pheromone distribution over the trails in CFM. However, since the ISM and the CFM indirectly correlate with each other, it might take too many iterations before satisfactory inputs can be found. In order to reduce the overhead of random sampling, we relax the requirement of satisfying the distribution of every branch along a virtual path in CFM to only satisfying the distribution of the last branch in the path that deviates from the target. In this way, we can significantly improve the efficiency of input selection, while simultaneously still able to guide the ants to make good decisions with respect to the goal.

5.3.3 Pheromone Update

During the incremental solution construction process in the ACO-based algorithm, each individual ant contributes its own partial solution as well as communicates with other ants by means of updating pheromones along the path it was traversing. The amount of pheromone laid down by an ant usually directly reflects the quality of this path with respect to certain goal. Since our goal is to reach a target node in the CFM, we perform quality assessment using a fitness function based on the characteristics of virtual paths being traveled in the CFM.

Given a virtual path $p_v=<b_{x0}, b_{x1}, \ldots, b_{xh}>$ in CFM(x_0=0), suppose CFM consists of M nodes, p_v misses the target node b_{goal} and b_{xq} ($1 \le q < h$) is the last node in p_v that is an ancestor of b_{goal}, the length of path segment from node b_{x0} to b_{xq} is called *valid length* of a virtual path. We compute the fitness value of p_v with the following formula:

$$fitness\,(p_v) = w_1 \times (M - levDis\,(b_{xq}, b_{goal})) + w_2 \times (\sum_{i=1}^{H} fitness\,(h_i)) / H + w_3 \times brchdiv\,(p_v) \quad (5.2)$$

In Formula (5.2), three factors are considered to influence the quality of a virtual path with respect to the target b_{goal}: 1) *level distance* between b_{xq} and b_{goal}, which indicates the length of the shortest path from b_{xq} to b_{goal}. The lower the level distance, the closer the ant is to the target and the higher the quality of the path. 2) *average good-quality potential* of p_v, a measurement of how well the ant follows the values of the routes granted by preceding ants in terms of the average pheromone intensity along the path segment in p_v till the last ancestor of the goal (i.e., from b_{x0} to b_{xq}). Each h_i represents a previously traversed path which is different from p_v but shares the same prefix as p_v from b_{x0} to b_{xq}. By taking an average on the fitness of all the valid h_i's, we are able to examine whether the current path p_v is getting closer towards the target in the right direction and how likely the p_v is to lead to traces with poor quality. In other words, a virtual path that can match more path segments with more valuable paths intends to have higher potential of good quality. 3) *branch diversity* of p_v: measures the overall degree of branch-taking diversity of all the conditional nodes (i.e., ITEs) along p_v. The $brchdiv(p_v)$ is a value proportional to the total number of *mismatched taken branches* between p_v and all previous paths h_i ($1 \leq i \leq H$). Given a conditional node C on p_v containing r possible branches, suppose p_v takes the branch $brch_j (1 \leq j \leq r)$, if there exists a path h_i which goes down through a different branch from $brch_j$ of C and at least one path h_k ($k \neq i$) which takes the same branch as p_v, the mismatched taken branch is increased by 1. Hence, the higher the mismatched taken branches, the higher the possibility that the path p_v is selected due to its quality estimation rather than branch stuck, which will be discussed in Section 5.3.5. We integrate the effect of the three factors to fitness evaluation by associating each with a different weighting coefficient.

The computed fitness value of p_v will then be used to update the pheromone trails along the segment from b_{x0} to b_{xq} in p_v. Besides pheromone accumulation, we also gradually reduce the pheromone intensities along the trails to help avoid unwanted convergence and explore new regions in the search space. Pheromones evaporate at a constant rate η. The

quality of a path would be degraded if it is not selected by enough number of ants and accumulated with enough amount of pheromone. Therefore, all valid edges along path p_v (i.e., from b_{x0} to b_{xq}) is updated with Formula (5.3):

$$phrm_{ij} = (1 - \eta)\, phrm_{ij} + fitness(p_v) \qquad (5.3)$$

In order to keep an updated correlation between the two sub-models, we will also update the pheromone intensities along the corresponding path in ISM that triggers p_v using $fitness(p_v)$.

5.3.4 Desirability Update

The desirability of an edge measures the attractiveness of an edge in building a solution. In general ACO applications, like Travelling Sales Problem, this value is usually fixed and set according to a priori knowledge, for example, the reciprocal of the edge length. However, the data and structure complexity of software programs make it difficult to tell the merit of an edge, e.g., a conditional branch, with respect to reaching the target block b_{goal} in advance. In this chapter, we update the edge desirability dynamically during the process of solution constructions. Before the algorithm starts, the $desire_{ij}$ of each edge in the ISM and CFM is initialized as follows:

$$desire_{ij} = \begin{cases} 0 & \text{if node } b_j \text{ belongs to the CFM and is not the ancestor of the } b_{goal} \\ 1 & \text{otherwise} \end{cases}$$

Intuitively, as the process of solution construction proceeds, different merits of edges can be distinguished by whether or not they are involved in past solutions, and how good quality the solutions are. After every iteration of solution construction, we update the desirability of each valid edge along the current traveling trace of the ant in both sub-models. Take the virtual path p_v=<b_{x0}, b_{x1}, …, b_{xh}> in Section 5.3.3 as an example, valid edges involve all the segments from b_{x0} to b_{xq} ($1 \le q < h$). Suppose p_v occurs at iteration H, given an edge <b_{xi}, b_{xi+1}> ($1 \le i < q$), its desirability is updated using Formula (5.4):

$$desire_{xi,xi+1} = \sum_k fitness(p_v^k) / W \qquad (5.4)$$

where W is the total number of traversed paths (i.e., past solutions) that contain $<b_{xi}, b_{xi+1}>$, $W<H$, and $\sum_k fitness(p_v^k)$ computes the total fitness value of these W paths. In other words, we compute the desirability of an edge by estimating its average fitness merit contributed by preceding solutions. Like the pheromone update, the corresponding path in ISM that triggers p_v is also updated with the new desirability value.

5.3.5 Daemon Actions

Although in the ACO, the indirect communication through the pheromone trails provides certain degree of global view of the whole graph for ant agents, it is still possible that the ants can get stuck in some local optimal solutions. In the case of software programs, the local optimum may occur especially when a dominant size of input space in the ISM leads to the same virtual paths in the CFM with inferior quality, since too frequent visits can still accumulate high enough pheromones along the paths that might misrepresent their values. In order to solve this problem, we apply centralized daemon actions over the solutions constructed by individual ants to adjust the pheromone deposit from a non-local perspective.

5.3.5.1 Local Optimum Detection

The local optimum checking is performed at an interval of K iterations. We check whether any form of local trapping happens among all the last K partial solutions in the CFM. In this work, we consider following two types of stuck:

1) *path stuck*:

Given K virtual paths: p_1, …, p_k, each of which has a valid length of l_1, …, l_k, respectively. Starting from the first node, we sweep all the paths node-by-node simultaneously to partition paths with the same prefix node sequence into groups. The sweeping stops until either the valid length of all the paths have been examined, or there exists a path group containing at least K/2 paths that share the same prefix node sequence

whose length is greater than its own valid length multiplied by ρ, a constraining factor usually greater than 0.5. For the latter case, we claim that a path stuck is detected.

2) *branch stuck*:

Suppose a total number of c different conditional statements are exercised at least once by the K virtual paths: p_1, …, p_k. Given a conditional statement containing R branches, we say it is *biased* if some branch i ($1 \leq i \leq R$) is executed at least once in more than K/2 virtual paths, and there exists another branch j ($1 \leq j \leq R$, $j \neq i$) in its counterparts, whose desirability is higher than that of the branch i. A branch stuck occurs if the number of biased conditional statements exceeds some threshold (e.g., c/2).

If any of the above two kinds of stuck is detected, we still need to distinguish whether the stuck is a true local optimum or it already represents a high-quality solution that we desire so far. Further judgment is performed by comparing the fitness of the new stuck paths or paths containing stuck branches with that of the previous stuck paths in records. If no fitness improvement is shown, we detect a local optimum and proceed to adjust the pheromones.

5.3.5.2 Additional Pheromone Adjustment

The goal of additional pheromone adjustment is to pull the stuck low-quality paths out of the trap through a centralized mechanism. In this step, we intentionally reduce the pheromone intensities along the valid edges of the stuck paths or on the stuck branches with less desirability. At the same time, the corresponding paths (i.e., input sub-domains) in the ISM are adjusted accordingly. By doing this, on one hand, the pheromone accumulation over these locations due to the frequent visits can be cut down; on the other hand, we implicitly increase the potential probability of those edges with higher quality to be chosen in the future search such that the stuck could be released.

The ACO-directed state space exploration repeats the above process of pheromone/desirability update and daemon actions, and stops with two possible outputs: either a valid input vector is found such that the target block has been reached or ants fail to reach the target within some pre-defined number of iterations. In this work, we focus

on those hard properties where the latter situation occurs predominantly. Note that in such cases, even though the target block is not reached, the eventual pheromone trails left by ants in the model still track down all their efforts made for property falsification while building up partial solutions incrementally. In particular, the pheromone trails would reveal promising areas in the search space where the solution is located.

5.4 Bounded Model Checking through Iterative Under-Approximation Based Program Abstraction

The new BMC algorithm based on the ACO-directed Under-Approximation based Program Abstraction (UAPA) is shown in Figure 5.3:

Algorithm *BMC_UAPA* (Program $\boldsymbol{P_{orig}}$, Assertion ***ast***, CFM ***cfm***)

1: P_{abs} = *constructAbstraction*(P_{orig}, *cfm*)
2: **do**
3: *res* = BMC_{orig}(P_{abs}, *ast*)
4: **if** (*res* = **UNSAT**)
5: *// absRefinement returns NULL if the input* P_{abs}=P_{orig}
6: P_{abs}= absRefinement(P_{abs}, *cfm*)
7: **while** (*res* = **UNSAT** and P_{abs} != **NULL**)

Figure 5.3 BMC Algorithm through Iterative UAPA

The first step of the algorithm is to construct an under-approximate program abstraction according to the pheromone trails left by ants in CFM. We choose multi-branch conditional statements (e.g., ITE, switch, while, etc.) as targets for abstraction since the semantics of these statements inherently divide the state space into multiple disjoint subspaces by each branch, making them natural candidates for search space reduction. Given a multi-branch conditional statement with R branches, correspondingly, it is represented as a multi-child node b_i in CFM with R outgoing edges $c_{i,j1}$, …, $c_{i,jR}$, each of which is associated with a pheromone vector $<phrm_{i,jk}, prob_{i,jk}>$ ($j_k \in \{j_{1,\ldots,}j_R\}$) by the end of the ACO process. As described in Section 5.3, the pheromone vector indicates how

good an edge is with respect to reaching the target node. Hence, we use some pre-defined *abstraction threshold* T_{abs} to suggest if a path should be included or excluded: if any $prob_{i,jk}$ is lower than T_{abs}, the corresponding branch in the statement linking from node b_i to node b_{jk} will be abstracted away. For example, suppose T_{abs}=0.3, given an ITE statement in the code:

if (*cond*) ***then*** {*then-body*}; ***else*** {*else-body*} (5.5)

where the *else-branch* has a probability of 85%, and *then-branch* is 15%. After abstraction, the ITE is transformed into:

assume(!*cond*); {*else-body*} (5.6)

where the then-branch is removed and the *assume*() statement restricts the search space at the ITE statement to only those states that satisfy the !*cond*, which are suggested by ACO as more promising to find solutions.

For the case of nested conditional statements, the abstraction decision on the outer-most layer statement would override that of the inner-ones. For instance, if the ITE example in (5.5) is located within a given branch of an outer conditional statement eligible to be removed, the entire inner ITE can be removed.

After the under-approximation is constructed, the abstract program P_{abs} is sent to a bounded model checker for assertion checking. If the BMC returns SAT, which means the property can be falsified in the reduced search space, the algorithm exits with the corresponding counterexample. Note that because the counterexample simultaneously satisfies all inserted *assume*() conditions along its path, this counterexample is also feasible in the original concrete program. If the BMC returns UNSAT, it may be caused by three possible reasons: 1) the property holds in the original program; 2) the P_{abs} is over-abstracted such that the property cannot be violated within the reduced search space; 3) the P_{abs} contains infeasible paths introduced by the conflicting assumption combinations from different abstract conditional statements. In any of the three cases, we need to perform abstraction refinement by restoring back some/all of the abstracted branches.

Given K multi-branch conditional statements that have been partially abstracted in the current P_{abs}, each of which is composed of two parts: abstracted portion (*abs*) and retained portion (*rp*). Every time we pick one statement s for refinement according to the following two rules: 1) no other abstracted conditional statements are nested within the body of s_{rp}; 2) s_{abs} is located closest to the target block among *abs* portions of K statements. Then, for any selected conditional statement s, we perform the following two-step refinement process:

step 1: Restore s_{abs} back while removing s_{rp}, then run BMC again on the resulting program;

step 2: If BMC in Step 1 still returns UNSAT, restore the entire statement s, and pick the next eligible conditional statement for refinement;

For instance, the two-step refinement for the ITE in (5.5) is as follows:

step1: *assume*(*cond*); {*then-body*}

step2: ***if*** (*cond*) ***then*** {*then-body*}; ***else*** {*else-body*}

where step 1 negates the assumption condition in (5.6) and step 2 restores the original ITE statement.

Partial abstraction may also be interleaved with the restoration process. At Step 1, if any conditional statements nested within s_{abs} have never been restored (i.e., they were fully abstracted) but satisfy the partial abstraction condition, instead of fully restoring s_{abs}, we attempt to restore such inner-statements partially only on their promising branches. Then, before moving to Step 2 for statement s, we apply the same two-step refinement for all the inner abstract conditional statements within s_{abs} until all of them have been fully restored or the BMC returns SAT. In other words, we carry out the refinement iteratively on each abstracted conditional statement in an inside-to-outside, bottom-up fashion, with BMC applied at each iteration. The refinement process stops either when the program is restored to its original status or a counterexample is found.

5.5 Experimental Results

Table 5.1 Bounded Model Checking Results with UAPA

Benchmarks	Abstraction Rate	Depth	Orig. BMC	Iterative UAPA	UAPA(Total)
Quicksort	3/16	5	73	29	**29**
		6	1683	743	**743**
		7	TO	2235	**2235**
Shellsort	1/5	25	530	187	**187**
		30	2439	2458	2458
		35	TO	9555	**9555**
Selection	2/15	5	34	6	**6**
		8	1480	165	**165**
		10	5792	1435	**1435**
Heapsort	2/9	25	64	8	**8**
		35	135	24	**24**
		45	521	30	**30**
kmp	1/5	40	500	127	**127**
		45	804	236	**236**
		50	1404	608	**608**
prog1	2/8	*13*	1963	979	**979**
		15	1651	62	**62**
		17	2062	411	**411**
prog2	6/19	5	798	33	**33**
		7	3610	244	**244**
		9	TO	1334	**1334**
pe_mut1	10/16	*NA*	2195	(1, 148)	**149**
pe_mut2	28/40		5612	(5, 1, 135)	**141**
pe_mut3	6/16		5155	134	**134**
pe_mut4	7/16		TO	(87,44,1884,120)	**2135**

(in sec, TO=10,000 sec)

We evaluated the proposed ACO-directed under-approximation based abstraction in bounded model checking with different software programs. As shown in Table 5.1, several sorting and selection algorithms as well as software applications were selected as benchmarks. The last four rows are benchmarks constructed from the policy engine module in a software defined radio system PSCR [72], which perform an equivalence checking between the policy engine and a mutated version of the code. For each

benchmark, given the assertion to be checked, we applied our ACO-directed concrete state space exploration with the goal of reaching the negation of the assertion. A total number of 150 ants (i.e., iterations) were tried for each case. The parameters used for probability computation in Formula (5.1) are set as α=1 and β=1. The runtime of ACO algorithm ranges from 5 seconds to a couple of minutes (last 6 benchmarks), it is relatively trivial compared to the extent of performance savings.

Based on the ACO results, an under-approximation of program is constructed with the abstraction threshold T_{abs}=0.6/R, where R is the number of branches of a conditional statement (R≥2 for *switch* statements). Table 5.1 compares the performance of the original BMC algorithm with the BMC through iterative UAPA for each benchmark. We use the C-Bounded Model Checker (CBMC) [13] as the underlying BMC engine, and all the selected properties are hard false properties. The second column in Table 5.1 shows the ratio of the number of abstracted conditional statements to the total number of conditional statements in the program at the last iteration of UAPA. As some of the benchmarks are loop intensive, the results of property checking under different unwinding depths are also reported.

From the results, the UAPA-based BMC resulted in dramatic performance improvement for almost all the cases (in bold) over the original BMC algorithm. For instance, in *pe_mut1*, the original BMC run took 2195 seconds. Our approach took two iterations (each value within the parenthesis represents the runtime at different refinement iterations). In the first iteration, only 1 second was needed due to the over-abstraction. After refining the model, 148 seconds were needed to find the counterexample. In *pe_mut3*, a speedup of 40 was achieved. For all benchmarks except for the policy engine (*pe_mut**) cases, our approach was able to disprove the property with no need of refinement. Even when iterative abstraction-refinement was needed, the total runtime of all the iterations was still much less than the original. These results show that the ACO-directed program abstraction is able to direct the UAPA towards promising areas in the code. The proposed iterative UAPA framework, thus, saves the time spent in exploring the unnecessary search spaces. The strength of ACO is further demonstrated in Table 5.2, where we compare the ACO-directed abstraction with a random-vector-directed

abstraction. The same iteration time and abstraction threshold are used in both techniques, as well as the same iterative refinement framework.

Table 5.2 Random-directed vs. ACO-directed Abstraction

Benchmarks	Abstraction Rate		Depth	Original	ACO	Random
	ACO	Random				
Quicksort	3/16	2/16	7	TO	**2235**	2568
Shellsort	1/5	1/5	35	TO	**9555**	9555
Selection	2/15	3/15	10	5792	**1435**	7162
Heapsort	2/9	1/9	45	521	**30**	245
KMP	1/5	1/5	50	1404	**608**	608
prog1	2/8	0/8	17	2062	**411**	(1,272,2062)
prog2	6/19	2/19	9	TO	**1334**	OM
pe_mut1	10/16	2/16	*NA*	2195	**149**	4614
pe_mut2	28/40	24/40		5612	**141**	569
pe_mut3	6/16	3/16		5155	**134**	(1,1304)
pe_mut4	7/16	10/16		TO	**2135**	(1,3545)

(in sec, TO=10,000sec, OM=Out of Memory)

It can be observed that ACO outperforms the random-vector-based execution in identifying good portions of the code for abstraction. Although running the program with random inputs was also able to find and prune away some highly biased branches in conditional statements according to the occurrence frequency, these branches do not necessarily link to the solution space. For example, considering benchmark *prog2*, the original BMC time was time out. With our ACO-directed abstraction, the run time was reduced to 1334 seconds; on the other hand, with the random-directed abstraction, the BMC performance was running out of memory while learning new clauses during the SAT solving, which is no better than the original case. This shows that, with the help of pheromone trails and daemon actions, ACO is able to narrow down the search to locations where the target property may be negated. The amount of pheromones accumulated during the iterative refinement reflects the quality of branches with respect to the verification target, and the daemon actions help jump out of local optimal solutions, while the random vector-triggered-execution usually get trapped in solutions

containing biased branches with higher execution frequency, which do not necessarily in connection with their values in abstraction and verifications.

5.6 Conclusions

In this chapter, we proposed an ACO-directed under-approximation based strategy to improve the performance of software model checking through search space reduction, especially for property falsification. An ACO-directed state space exploration was applied trying to construct a solution that falsifies the property through the coordination among ants. The ants leave pheromone trails along program structure to indicate the promising areas in the code from which an under-approximate model is built. Centralized daemon actions are also adopted to assist the ants to avoid local optimum solutions. A new BMC algorithm through the proposed UAPA framework is presented to search for solutions over the reduced state space, which is iteratively refined until it contains enough details to verify the property. Experimental results showed that with the proposed technique, the performance of software BMC for property falsification can be improved significantly, reaching up to 40 times of speedup. The ACO also demonstrated effectiveness in guiding the code abstraction over random-vector-directed abstraction.

Chapter 6

Constraint-Reduction-Based Abstraction with Single Mined Invariants

6.1 Introduction

The abstraction technique proposed in Chapter 5 reduces search space through under-approximate model built over the promising program areas, hence, it is more applicable to property falsification rather than property verification, as the latter case requires restoring all the removed structure back to its original location before a verification conclusion can be made. In such a context, verification won't show much promise in performance saving than the original. In this chapter, we explore hybrid approaches along another direction of the general abstraction technique – over-approximation, which make them more suitable for proving the properties.

We propose a new CEGAR framework integrated with a new invariant-directed code-level constraint-reduction-based abstraction (CRA) techniques with the goal of enhancing the scalability of software bounded model checking. The proposed CRA is an abstraction technique that builds an over-approximate model by removing some irrelevant constraints or state variables in the symbolic transition relation with respect to the verification of the target property. As introduced in Section 2.3.1.1, the conventional constraint-reduction-based abstraction technique generally reduces system complexity via simply removing

the existing variables or constraints, which may lead an initially too coarse over-approximate model and thus results in more refinement steps. Instead, we leverage invariants to further confine the state space by retaining the essential correlations between the relaxed variables, while removing the corresponding statements(or constraints) used for variable assignment at the code-level. As a result, on one hand, the reduced constraints significantly mitigate the program complexity; on the other hand, the added invariants help improve the precision of abstraction such that the number of refinement iterations can be decreased accordingly. Finally, we employ a CEGAR process on the over-approximate model using assume-guarantee based model checking. If it fails and the counterexample is found spurious, refinement will be performed by gradually restoring the removed constraints until the property is verified.

In particular, we discover these invariants via data mining over the profile data collected from dynamic program execution. We take one step beyond purely adding the invariants to constraint the state space as in the work in Chapter 4, where the potential performance savings it could bring may largely rely on how effective the underlying model checker utilizes the added constraints during the property verification process. In this chapter, the invariants are used to further construct over-approximate abstract spaces by removing the related program code, so that the complexity of the system can be significantly reduced and the underlying model checker can directly use the information provided by the invariant without wasting efforts to re-reasoning about it. Our mining approach not only overcomes the scope and granularity limitations in existing invariant detection technique [17], but it also extends the application of invariants beyond predicate abstraction [7]. Moreover, the invariant mining and validation in our approach is a one-pass process before the property verification takes place, whose results can be reused for verifying multiple properties for the same program. Therefore, it also overcomes the limitation of conventional CRA techniques that have to conduct the entire abstraction process all over again for different property verification. Experimental results demonstrated that our new code-level CRA abstraction directed by invariant mining can achieve up to several orders of magnitude of performance improvement over the original software bounded model checking.

The remainder of the chapter is organized as follows. Section 6.2 presents the code-level constraint-reduction-based abstraction using verified single invariants selected by the mining algorithm. The CEGAR framework for software bounded model checking built with the CRA abstraction is described in Section 6.3, followed by experimental results in Section 6.4. Finally, Section 6.5 concludes the chapter.

6.2 Constraint-Reduction-Based Abstraction with Single Mined Invariants

6.2.1 Potential Single Invariant Mining

In order to apply constraint-reduction-based abstraction, we employ the property mining algorithm described in Chapter 4 to discover potential variable-level invariants. We focus on Single Linear Properties(SLP) mining in this context, which describes the basic comparison relation ● between two candidate variables v_1 and v_2 held at certain code line k (i.e., after the line k is executed), denoted as:

$$\wp :: k \Rightarrow v_1 \bullet v_2, \quad \bullet \in \{>, \geq, <, \leq, =, \neq\} \tag{6.1}$$

6.2.2 Constraint-Reduction-Based Abstraction over Single Invariants

The conventional constraint-reduction-based abstraction (CRA) for programs refers to relaxing the correlation among variables such that the complexity of the program can be reduced. Furthermore, we conduct the CRA over an invariant property by removing the original statement constraints on those variables involved in the property, while at the same time inserting the property as a new constraint for state space restraining. Note that the properties learned from the above association rule mining algorithm are just potential invariants that hold over the execution data. In order to be used as constraints for CRA, a validation process is applied to establish the truth of each potential invariant.

Let $P(v_1, v_2)$ in the form of $\wp :: k \Rightarrow v_1 \bullet v_2$ be such a verified SLP invariant, where $\bullet \in \{>, \geq, <, \leq, =, \neq\}$ represents a linear relation ● held between variables v_1 and v_2 after executing the statement at line k through all the possible execution paths. As Figure 6.1

shows, the construction of CRA over $P(v_1, v_2)$ consists of the following three steps: Firstly, we search the statements backward in the original code (Figure 6.1(a)) along the data flow and identify the locations of the last possible assignment statements for v_1 and v_2 (line l and line k, respectively, in this example). Next, release all the correlated constraints with respect to these variable assignments. For example, in Figure 6.1(b), the constraints defined by $func_1$ at line l and $func_2$ at line k are removed and both variables v_1 and v_2 are set free so that they can take any value. Accordingly, any variable involved in the parameters of $func_1$ and $func_2$ (e.g., y, z in Figure 6.1(a)) whose purposes are only for the definition of v_1 at line l or v_2 at line k, can be set free. The above steps can be repeated until no further variables can be removed. Finally, we using the *assume*() statement to insert $v_1 \bullet v_2$ as a constraint in the abstracted code (Figure 6.1(b)) at the location corresponding to the line k in the original program. This helps to constrain the two variables to have some relations similar to the original code.

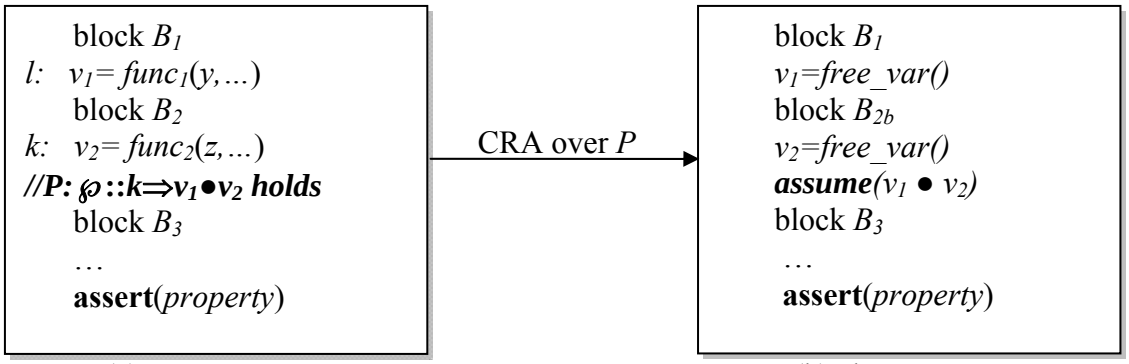


Figure 6.1 Constraint-Reduction-Based Abstraction over P

According to the definition of SLP, the invariant $P(v_1, v_2)$ is actually an implication deduced from preceding constraints on v_1 and v_2. Take again the code segment in Figure 6.1 as an example, the following formula holds:

$$\forall v_1 \forall v_2 (\Psi \wedge v_1 = func_1(...) \wedge v_2 = func_2(...))$$

$$= \forall v_1 \forall v_2 (\Psi \wedge v_1 = func_1(...) \wedge v_2 = func_2(...) \wedge P(v_1, v_2))$$

$$\subseteq \forall v_1 \forall v_2 (\Psi \wedge P(v_1, v_2))$$

where Ψ represents the conjunction of preceding constraints that define all the arguments of $func_1$ and $func_2$. Hence, by replacing the left-hand side constraints with the right hand side property, the abstracted code obtained by CRA over P is an over-approximation of the original program behavior. The above method can easily be extended to multiple invariants.

6.2.3 Potential Single Invariant Selection

Apparently, with more invariants to be employed for CRA, on one hand, the code complexity can be more greatly decreased, however, on the other hand, with more constraints being removed, the abstracted model might get too coarse to hold sufficient information to verify the target property, which in turn may introduce more iterations of refinement and degrade the potential performance saving brought by abstraction. In addition, different invariants can result in different sets of constraints to be released, which is another factor that may influence the effects of CRA with respect to the property verification. Therefore, tradeoffs have to be made to balance the number and the power of invariants getting involved in the abstract model and the complexity of the resulting code.

We select valuable properties out of all the mined invariants by considering the combination of three factors. The first two factors are the statement distance and the variable distance. The former represents the distance between the location of the learned invariant and the location of target property, and the latter counts the total level differences in the data dependency graph between every pair of a variable involved in the invariant (e.g., v_1 and v_2 in Figure 6.1 for $P(v_1, v_2)$) and a variable involved in the property; The closer the two distances, the higher the correlation between the invariant and the verification of the target property might be, hence, the more valuable the invariant is considered. The third factor is the operation complexity, which estimates the total computational complexity estimation of all the operations involved in the corresponding constraints (e.g., $func_1$ and $func_2$ in Figure 6.1 for $P(v_1, v_2)$) that would be removed if the invariant is selected for CRA. The complexity computation also needs to take the data types of the corresponding operators into account. For instance, the

multiplication of two double-typed variables should be more expensive than multiplying two integer variables. Since this factor directly reflects how much complexity reduction the invariant can bring to the code when it is applied to CRA, the higher the complexity, the more powerful the invariant would be.

We assign different weight coefficients to each of the above factors and evaluate the value of an invariant in terms of the property verification as an integrated effect of the three, where the factor two and three would be considered as more crucial. Eventually, a limited number of the resulting top-rated invariants will be selected to conduct CRA.

6.3 CEGAR with Constraint-Reduction-Based Abstraction

In this section, we introduce how the single-invariant-directed CRA can be fitted into the conventional CEGAR framework for model checking through iterative abstraction and counter-example guided refinement scheme.

6.3.1 CEGAR-CRA Framework

Algorithm *CEGAR-CRA*(***M_{orig}***, ***prop***, ***Inv***)

```
1:    M_abs = Build_CRA(Inv)
2:    Res = Model_Checking (M_abs, prop)
3:    if (Res = SAT)
4:        if (M_abs != M_orig and Res.cex is spurious)
5:            M_abs' = Cex_Guided_Refinement(Res.cex, M_abs)
6:            M_abs = M_abs'
7:            Goto line 2
8:        else return Res.cex          // property prop is falsified
9:    else return verified         // if Res=UNSAT
```

Figure 6.2 CEGAR-CRA Framework

Figure 6.2 shows the pseudo code of the CEGAR framework based on invariant-directed CRA. Given the original program M_{orig} and the target property to be checked *prop*, let *Inv*

denote the set of invariants obtained through the above mining and selection algorithms and passed the validation. We first construct CRA over the *Inv* set using the approach described in Section 6.2.2 and obtain an abstracted program M_{abs} (line 1) with all the *Inv*-related constraints removed. The abstract M_{abs} is then model checked against the *prop* to see whether it is satisfied or not (line 2) using a conventional software bounded model checker. If the result *Res* is UNSAT (line 9), we exit the algorithm by reporting that the *prop* holds in both the abstraction and in the original program since M_{abs} is an over-approximate model. Otherwise, if model checking fails (i.e., *Res* is SAT), a counterexample *Res.cex* will be returned illustrating how the *prop* is violated in the abstracted model M_{abs}. The abstract counterexample could include abstract transition actions which may not find a matching counterpart in the concrete model due to the over-approximation. Therefore, when M_{abs} remains an over-approximate model over M_{orig}, we perform spuriousness checking on the abstract *Res.cex* to see whether a corresponding concrete counterexample exists in the concrete model. If yes (line 8), *prop* is falsified with the returned *Res.cex*. Otherwise (line 5-7), the *Res.cex* is spurious and refinement is conducted to strengthen the M_{abs} by restoring some of the previously removed constraints back. The above steps are repeated iteratively on the more precise abstract model until either the *prop* is proved or falsified.

6.3.2 Counter-Example Guided Refinement

Suppose the original program M_{orig} contains N input variables: $V_C=\{v_{C1},\ldots,v_{CN}\}$, let $D_{C1},\ldots,D_{CN}$ represent the corresponding input domains for the N inputs, respectively, where all the values of V_{Ci} fall in the range of D_{Ci} ($1\leq i\leq N$). According to the above invariant-directed CRA algorithm, the abstract program M_{abs} retains the original N input variables, but defined over abstract input domains $D_{A1},\ldots,D_{AN}$, respectively, where $D_{Ai}\supseteq D_{Ci}$, we denote these abstract input variables as $V_A=\{v_{A1},\ldots, v_{AN}\}$. In addition, suppose R intermediate variables: $x_{A1},\ldots, x_{AR}$ defined at lines $l_1, \ldots., l_R$, respectively, are set as free variables during the abstraction process, denoted as $x_A=\{x_{A1}@l_{X1}, \ldots, x_{AR}@l_{XR}\}$. They can now be viewed as R additional pseudo input variables, each of which falls in the domain $D_{Xj}(1\leq j\leq R)$ defined by its own data type. An abstract counterexample $cex=\{V_{in}, L_A\}$ is composed of two elements: 1) V_{in}: an input vector with $N+R$

values:[v_{A1},…,$v_{AN,}$,x_{A1}@l_{X1},…, x_{AR}@l_{XR}], where $v_{Ai}\in D_{Ai}(1\le i\le N)$, $x_{Aj}\in D_{Xj}$ $(1\le j\le R)$ and 2) L_A: a sequence of program locations: (l_{A1}, …, l_{AM}) representing the program execution trace correspondingly, when V_{in} is applied as the program input.

Since the CRA abstraction only expands the variable domain instead of changing the count of the original input variables, the spuriousness checking of an abstract *cex* comes down to a one-pass execution of the original program by running the original program M_{orig} with the vector <v_{A1},…,v_{AN}> as its inputs. Suppose the corresponding concrete program execution trace is L_C=(l_{C1}, …, l_{CP}), the *cex* is spurious if any of the following conditions is satisfied:

i) $v_{Ai}\notin D_{Ci}$ for any $1\le i\le N$

ii) (l_{C1}, …, l_{CP}) != (l_{A1}, …, l_{AM}) and *prop* is not reached

iii) *prop* is reached but not violated

The first condition violates the concrete domain of input variables, and the last two cases could not falsify *prop* since the pseudo input variables take values different from those in the V_{in} after executing the concrete constraints that have been removed during the abstraction, which lead to different execution results. Therefore, the abstract *cex* does not match to a concrete one.

Algorithm *CEX-Guided_Refinement*(***cex, M_{abs}, S_T***)	
1:	$M_{abs}'=M_{abs}$ + s_t@l_{St} *in* S_T
2:	**if** (sat_check(M_{abs}'((V_{in} \(remaining_free_vars))$_{cex.Vin}$))=**UNSAT**)
3:	**return** M_{abs}'
4:	**else**
5:	$t = t - 1$
6:	$M_{abs}=M_{abs}'$
7:	**Goto** step 1

Figure 6.3 CEX-Guided Refinement Algorithm

The spurious abstract *cex* indicates that the abstraction is too coarse and needs to be refined. Suppose $P(v_1, v_2)$: $\wp::k \Rightarrow v_1 \bullet v_2$ is a verified SLP invariant used for CRA, correspondingly, during the abstraction, a set of statements $S_T=\{s_1@l_{S1}, \ldots, s_t@l_{St}\}$ has been removed, where $l_{S1}<\ldots<l_{St}<k$, we apply a counterexample-guided abstraction refinement algorithm to gradually add back some of the removed constraints in S_T so that the spurious *cex* can be eliminated. In particular, we focus on removing the spuriousness caused by the conflict value assignments among those intermediate free variables involved in the invariants used for CRA (e.g., v_1 and v_2).

As shown in Figure 6.3, starting from the statement in S_T that is located closest to k, i.e., $s_t@l_{St}$, one statement is restored back to the abstract model M_{abs}. Simultaneously, the corresponding variable defined by this statement is taken out from the current free variable set (line 1). The resulting enhanced abstract model is denoted as M_{abs}'. It can be easily proved that M_{abs}' contains a subset of behavior over M_{abs} with more constraints added. Let the spurious $cex=\{V_{in}, L_A\}$, a constraint formula is built by conjoining all the existing statements in the M_{abs}', with all the existing input variables(original and pseudo) instantiated with values in V_{in}, except for those abstract variables that remain free.

An example of M_{abs}' is given in Figure 6.4 where internal variables $x_A=\{y@l_{S1}, z@l_{S2}, v_1@l_{S3}, v_2@l_{S4}\}$ are abstracted as free variables in the initial M_{abs}, and the M_{abs}' has already restored the statements s_3 and s_4, while y and z on which the definition of v_1 and v_2 are dependent remain free.

```
                …
s1@lS1:    y = free_var()          // remains free_var
s2@lS2:    z = free_var()          // remains free_var
s3@lS3:    v1 = func1(y,… )        // restored in Mabs'
s4@lS4:    v2 = func2(z,… )        // restored in Mabs'
           assume(v1 > v2)
```

Figure 6.4 A CEX-Guided Refinement Example for CRA

Given a spurious *cex*, where $v_1@l_{S3}=val_1$ and $v_2@l_{S4}=val_2$, the constraint formula will be constructed as follows:

$$M_{abs}'(V_{in}\backslash\{y@l_{S1}, z@l_{S2}\})_{cex.Vin} = M_{abs}'(V_A \cup x_A\backslash\{y@l_{S1}, z@l_{S2}\})_{cex.Vin}$$

$$= M_{abs}'(V_A)_{cex.Vin} \cap (v_1@l_{S3}=val_1) \cap (v_2@l_{S4}=val_2) \quad (6.1)$$

With the constraint formula, we then perform satisfiability check on the formula to verify whether the enhanced model M_{abs}' is precise enough to eliminate the spuriousness. If the formula returns UNSAT, the M_{abs}' is returned as the refined abstract model since the same instantiation of non-free variables as in V_{in} is no longer a valid input vector to the M_{abs}', thus, the *cex* has been removed (line 3). Otherwise, the corresponding solution to the formula means that even with the added statement, an appropriate assignment on the rest free variables can still make the *cex* a possible input case in M_{abs}'. We repeat the above process and attach more statements back to the model (line 5-7) until either all the statements have been restored or the satisfiability check returns UNSAT.

Considering the example in Figure 6.4 again, if the formula in (6.1) is satisifiable, add back the statement $s_2@l_{S2}$ and repeat the process, otherwise, the M_{abs}' becomes the refined abstract model for next iteration.

The above refinement algorithm induces extra overhead of satisfiability checking every time when examining whether to restore a statement or not; however, since the checking is built on a specific spurious counterexample in which most of the variables are instantiated with real values, only a limited number of variables is left unassigned, the overhead of refinement is thus relatively small compared to the property verification, as can be demonstrated in the experimental results.

6.4 Experimental Results

We conducted model checking experiments with several C program selected as benchmarks, including both the academic algorithms as well as real-world software applications (the policy engine module in a software defined radio system PSCR [72]). For each benchmark, the association rule mining for SLP (s_{th}=0.8, c_{th}=1.0) is applied to

discover potential invariants over the profile data, which is collected via dynamic program execution either for N=100 times on randomly generated inputs or when the transaction count in the mining database exceeds 5000, whichever comes first. All the potential invariants were then validated before selected as constraints for abstraction with respect to the target assertions to be checked. For assertion verification, we used CBMC (C Bounded Model Checking) [6] as the base model checker and only safety properties were considered.

Table 6.1 Bounded Model Checking Results with Single Invariant-Directed CRA

Benchmark	Depth	Orig. BMC	Abs. Ratio(Kilo)	Constraint-only	CRA Abs. & Refinement
ShellSort	15	TO	196.2/372.3	TO	**2**
	18	TO	283.3/591.0	TO	**8**
	20	TO	351.5/772.9	TO	**3**
BubbleSort	10	3483	26.5/78.0	4138	**3**
	15	TO	53.4/178.4	TO	**262**
	20	TO	88.6/322.1	TO	**804**
KMP	15	625	505.4/632.3	765	**397**
	18	1307	658.2/845.0	1184	**922**
	20	3682	769.3/1002.8	1504	**3425**
BinarySrch	15	2408	505.4/632.3	2670	**24**
	18	TO	658.2/845.0	TO	**66**
	20	TO	769.3/1002.8	TO	**61**
PolicyEng	25	1006	3344.0/3566.2	869	5,5,869

(in sec, TO=10,000 sec)

The bounded model checking results for different benchmarks under different unwinding depths are shown in Table 6.1, as most of the benchmarks are loop-intensive. Whenever refinement is needed, the performance at every refinement iteration is listed, separated with comma. In addition, we also provide the abstraction rate of each benchmark, which is calculated as the ratio between the clause count (in kilo) of the abstracted model (at 1st iteration) and that of the original model.

From the results we can see that with the CRA, several orders of magnitude of performance improvement in BMC can be achieved over the verification of the original code, with the size of the model being reduced to up to 1/5 of the original. For instance, the *shellsort* timeout for all the three unwinding depths with the original BMC, while with over-approximation based on the learned invariant, we are able to cut the original model down to around half size since the invariant is located within the major loop. As a result, only less than 10 seconds (> 1000x speedup) is needed to obtain the verification results. The results obtained demonstrate the power of invariant-directed CRA not only in reducing the program complexity but also in retaining essential relations among variables needed for proving the assertions such that the savings can extend beyond the degree of size reduction.

It can also be observed that for almost all the cases, the first-pass abstraction is already sufficient to prove the property with no refinement. It further indicates the high level of precision and efficiency of our potential invariant mining algorithm in finding crucial invariants, hence, the added invariants can greatly reflect the essence of the correspondingly relaxed constraints with respect to the property verification without introducing unwanted additional information to the state space. In the cases when refinement is needed (e.g., last benchmark), the total verification performance over different iterations can still bring adequate savings upon the original one. The cex-guided refinement process goes fast enough (< 5 sec) such that the overhead can be regarded as negligible.

We also compared the results against the mining-directed constraint based exact abstraction approach from Chapter 4 where the mined true invariants are added as additional assumptions to the original code with no over-approximations involved. From the results we notice that by adding only invariants, it would achieve limited success among all the benchmarks. In many cases, the performance is even worse than the original, because the added invariants can sometimes increase the size of the model, which may incur extra burden to the reasoning process. The worst case of our approach happens when the refinement eventually requires restoring all the relaxed statements back, in which case our approach would results in comparable performance with the

constraint-adding approach since the falsification iterations usually take short time (e.g., the final benchmark *PolicyEng*).

6.5 Conclusions

In order to enhance the scalability of software model checking, we propose a new CEGAR framework incorporated with a new code-level constraint-reduction-based abstraction technique. It utilizes the invariants discovered through data mining to significantly reduce the program complexity by freeing those less relevant constraints, while at the same time, retaining the essential correlations among relaxed variables. Experimental results demonstrated the effectiveness of our approach in improving the performance of software model checking, as well as outperforming the previous work where invariants are not used for over-approximation.

Chapter 7

Local Invariant-Directed Path-Level Partitioning with CRA

7.1 Introduction

The invariant-directed CRA technique proposed in Chapter 6 is constructed over the invariants in the form of Single Linear Properties (SLP) that are discovered to hold *globally* over the entire state space at their program locations. Note that, by *global* SLP we refer to the truth rather than the location of the SLP in the code. For example, an invariant within a loop is a global SLP as long as its truth does not depend on conditions other than the condition of the loop it belongs to. Therefore, the applicability of invariant-directed CRA highly depends on how many such global invariants can be mined, and how powerful they are in terms of program complexity reduction and property verification once being used for CRA. Such limitation can probably lower the potential savings that might be brought by the single-invariant-directed CRA.

Figure 7.1 shows a motivating example. Given the original program in Figure 7.1(a), suppose two invariants are mined using the mining algorithm proposed in Chapter 4, both are complex property implications (CPI) spanning over different program locations. By analyzing the two invariants, we find that the only possible global SLP is $\{prop_2\}$ that holds at the end of the execution of *else-body*$_1$ (i.e., after line 5 is executed), since the

antecedent of inv_2 implies the else-branch of the first if-then-else (ITE) to be taken. Figure 7.1(b) shows the corresponding abstract program constructed over $prop_2$ using the invariant-directed CRA technique introduced in Chapter 6, assuming that the entire *else-body*$_1$ can be replaced by $prop_2$. We can see that, with the preceding CRA approach from Chapter 6, the program complexity reduction is just limited locally to else-branch$_1$, with all the other portions of the code preserved. Although $\{<\wp::7>\&(prop_1)\}$ is also involved in the invariant set, it is not globally true, thus, no single-invariant-directed CRA can be applied for inv_1.

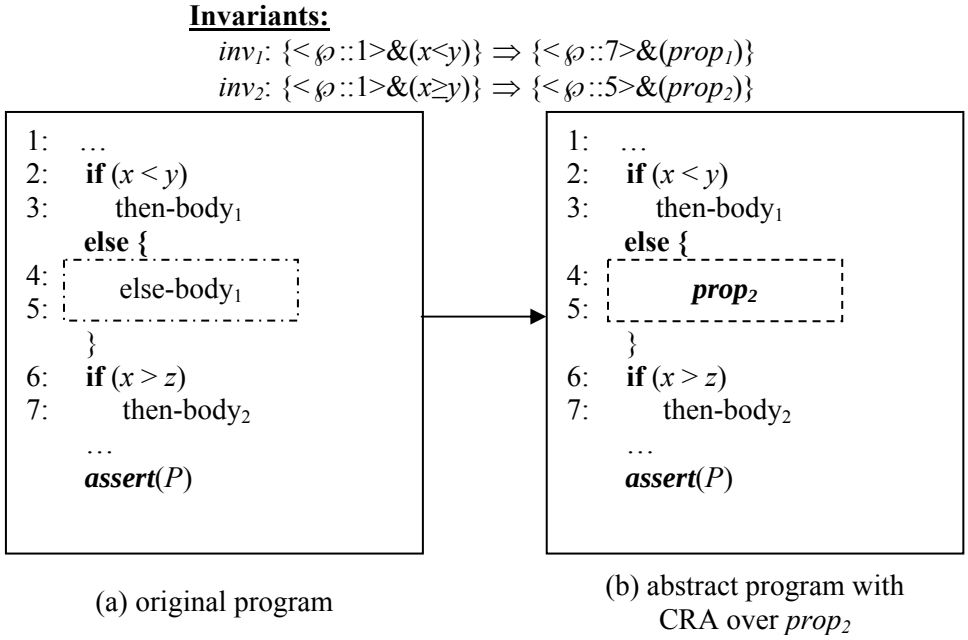


Figure 7.1 A Motivating Example of Partitioned CRA

From the example in Figure 7.1 we can see that, due to the complexity and size of software programs, treating the entire state space of a program uniformly as an abstraction unit may to some extent limit the applicability as well as the efficiency of invariant-directed CRA. This motivated us to seek the partitioning of the state space into disjoint sub-spaces, and keeping abstraction (may be different) locally to each sub-space to fully utilize the power of CRA abstraction. According to the characteristics of software programs, it is natural to use control flow (i.e., paths) as the means for state space partitioning. For example, re-examining the invariant set in Figure 7.1, we can find that

the antecedents of both invariant implications, if taken, actually lead to two different program execution flows:

inv_1: $\{<\wp::1>\&(x<y)\}$ → path 1-2-3-6…

inv_2: $\{<\wp::1>\&(x\geq y)\}$ → path 1-2-4-5-6…

In this chapter, we propose a path-level partitioned abstraction technique under the guidance of mined invariants in the form of complex property implications. It extends the previous invariant-directed CRA abstraction proposed in Chapter 6 by localizing different abstractions within each functionally disjoint partition based on the invariant inferred only in the partition. Moreover, in order to maximize the strength of the partitioned abstraction, we further improve the previous mining algorithm built on random inputs to coverage directed inputs such that the underlying mining database can gain a more balanced occurrence frequency over different branches. Finally, we propose a new CEGAR paradigm built on the partitioned abstraction technique for bounded model checking of software.

The partitioned abstraction further enhances the power and flexibility of the CRA abstraction from the following three perspectives. Firstly, since the partitioning is based on the control paths, each partition itself has already removed some of the program logic, e.g., only containing the then-branch of an ITE statement, consequently, greater savings could be expected when abstraction is applied on the partition with a reduced complexity rather than the entire code. Secondly, the partitioned abstraction no longer requires the mined invariants to be global, locally implied invariants can also be used for abstraction, which increases the applicability of the invariant-directed CRA on broader program areas. Thirdly, since the partitions correspond to sub-state-spaces that are mutually disjoint, different abstraction strategies can be conducted and refined within each individual sub-space independently; hence, it provides a flexible framework for abstraction selection. In addition, as each partition can be viewed as an under-approximation of the code, the reduced program enables potential performance savings for the case of property falsification as well. Experimental results on a set of real-world programs demonstrated that with the partitioned abstraction technique, drastic performance improvement can be

achieved compared to the original model checking in both property verification and refutation. The proposed technique also exhibits great potential of outperforming the global SLP invariant based CRA abstraction in most of the experiments.

Partition and abstraction have been two popular techniques in the area of model checking. Sebastiani et al. in [92] proposed a similar concept of combining property-driven partitioning with localized abstraction for LTL model checking, the state space is partitioned according to the states of the property automaton. Our work applies to software domain, and differs from [92] in the partitioning and abstraction strategies. In the context of software verification, both [34] and [93] explored localized predicate abstraction such that different levels of precision were maintained in different parts of the program, and abstraction is refined locally to some traces, or to some control locations[93]. However, no explicit partitioning on the program structure was involved, and the localization was passively determined by the selection of predicate abstraction. Instead, we propose explicit integration of partitioning and the invariant-directed CRA abstraction, abstraction in one partition is independently constructed and refined from the other functionally disjoint partitions, which enables the flexibility of applying non-uniform abstraction. This also allows us to find more invariants as we focus on a smaller state space within one partition as opposed to the entire program.

The remainder of the chapter is organized as follows, In Section 7.2, we presents the CPI directed path-level partitioned CRA scheme. The corresponding CEGAR framework for software bounded model checking built with the partitioned CRA is described in Section 7.3. Section 7.4 discusses the experimental results. Finally, we conclude in Section 7.5.

7.2 CPI-Directed Path-Level Partitioning with CRA

7.2.1 Potential Complex Property Implication Mining

In order to apply the partitioned CRA, we employ the property mining algorithm described in Chapter 4 to discover potential variable-level invariants. Unlike the focus on mining SLPs in Chapter 6, Complex Property Implications (CPI) are also mined in this case, which describes the implication relations between two linear properties spanning

across different program locations. Since we partition the program based on the control paths after mining, the original definition of the CPI in Formula (4.3) is slightly extended into Formula (7.1).

$$\{<\wp::\lambda_1> \,\&\, cond(x_1, ..., x_n)\} \Rightarrow \{<\wp::\lambda_2> \,\&\, (y_1 \blacklozenge y_2)\} \quad (7.1)$$

The new CPI indicates that if some linear relation $cond(x_1, ..., x_n)$ holds at line λ_1(i.e., righter after the statement at located line λ_1 is executed) of the program, then $y_1 \blacklozenge y_2$ is definitely true at line λ_2, where $x_1,\ldots, x_n$, y_1, y_2 are program variables or constants, $\lambda_1 < \lambda_2$, and $\blacklozenge$ is one of the comparison operators from set $\{<, \leq, ==, !=, >, \geq\}$. The $cond(x_1, ..., x_n)$ in (7.1) is a logical function of $x_1, \ldots, x_n$, referring to the conditional expression or the negation of the conditional expression of a certain control statement in the program, like ITE, switch, while, etc. In this chapter, we will use ITE statement as an example for partitioning. Note that, the mined CPIs are also just potential invariants; they need to be validated before being used for partition and abstraction.

7.2.1.1 Branch Coverage Directed Dynamic Program Execution

So far, the underlying mining database is constructed using the profiled data generated by dynamic program execution over randomly generated input vectors. Although random execution has its benefits, it could also result in biased visits over certain program structure, e.g., some branches are disproportionally visited. If the program structure is reached by a dominant range of inputs, it may consequently influence the quality of the mined invariants in terms of how useful they are in helping with the program abstraction and assertion verification. For the case of path-level partitioned CRA, due to the inherent feature of the association rule mining, only when branch execution frequency of an ITE is balanced enough, can the mining algorithm be able to learn different CPIs with antecedents covering both *if-then* and *if-else* branches. Therefore, we extend the CPI mining over selected inputs directed program execution profiles with the goal of mitigating the biased coverage among the branches. The problem can be defined as follows: Let S_{ITE} be a candidate ITE statement in the program containing two branches: $brch_{bias}$ and $brch_{tgt}$, suppose a dominant range of inputs would lead the program execution flow down through the $brch_{bias}$, we apply a meta-heuristic algorithm combining

a local-search-based input generation approach with stochastic input model, trying to select appropriate input vectors to build the mining database such that the both branches $brch_{bias}$ and $brch_{tgt}$ occur in a relatively balanced number of transactions.

Given a program with N input parameters, I_1, …, I_N, each of which is defined over domain D_1, …, D_N, respectively. We divide each domain D_i equally into R_i disjoint sub-domains, denoted as D_{i1}, …, D_{iRi}, in ascending order. Figure 7.2 shows the stochastic model of program inputs. For each sub-domain D_{ij} of input I_i, we assign a weight factor w_{ij} estimating the desirability for the value of I_i to fall into the range of D_{ij}. Initially, $w_{ij}=W_{init}/R_i$ for any possible j, where W_{init} is a pre-defined weight value.

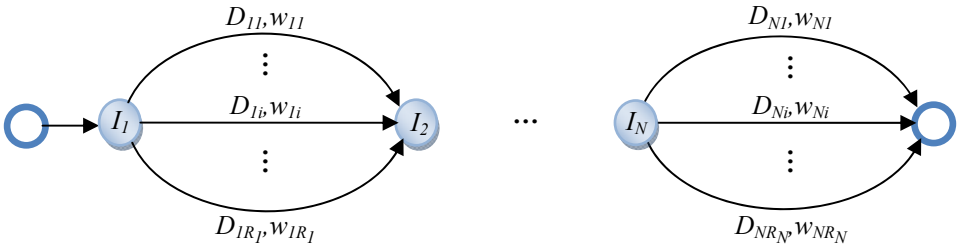


Figure 7.2 Input Model for Branch Coverage Directed Input Generation

Based on the input model, a stochastic input vector generation process is conducted as follows: for every input I_i, we first use the Monte-Carlo method to stochastically pick a sub-domain D_{ij} according to the weight distribution over all its R_i sub-domains, the larger the weight, the higher the probability this sub-domain is picked. Then, the value of I_i, denoted as val_i is randomly sampled within the range of the picked D_{ij}. The resulting generated input vector $V(val_1, ..., val_N)$ is then fed into the program for execution. However, in our extended mining algorithm, not all the program execution profiles are stored into the mining database. Periodically, during the mining database construction, we enter the *branch coverage enhancement* stage if the visiting frequencies between the two $brch_{bias}$ and $brch_{tgt}$ are overly biased, which means the ratio between the two frequencies exceeds certain threshold $balance_{thd}$ (line 1, Figure 7.3). Within this stage, we apply the Local-Search-Based Input Generation algorithm as shown in Figure 7.3 to

attempt to restore the balance, and only valid execution profile that can improve the branch coverage are preserved.

Algorithm *Local-Search-Based Input Generation* **($brch_{bias}$, $brch_{tgt}$)**

1: **while** (($freq(brch_{tgt})/freq(brch_{bias}) \leq balance_{thr}$) **and** within iteration limit)
2: Run program with stochastically generated input vector $V(val_1, \ldots, val_N)$
3: **if** ($brch_{tgt}$ is missed)
4: **for** each related input I_{rk}
5: **while** ($brch_{tgt}$ is not reached **and** within iteration limit)
6: $brchdist = expressionEvaluation(brch_{tgt})$
7: $w_{rk}[] = adjustWeight(I_{rk}, brchdist)$
8: $\Delta = compuDelta(branchdist, w_{rk}[])$
9: $V = localSearch(V, I_{rk}+\Delta)$
10: Run program with input vector V
11: **if** ($brch_{tgt}$ is reached)
12: Add execution results to the mining database
13: **end while**
14: **for** each related constants $const_{rk}$ of I_j
15: repeat step 5 – 13, $V=localSearch(V, const_{rk}$ or $const_{rk}+\Delta)$
16: **end for**
17: **end for**
18: **end while**

Figure 7.3 Local-Search-Based Input Generation Algorithm

Suppose M input variables $I_{r1}, \ldots, I_{rM}$ ($M \leq N$, $\{r_1, \ldots, r_M\} \subseteq \{1, \ldots, N\}$) are related to the conditional expression of the target ITE statement S_{ITE} , i.e., the evaluation of the expression are dependent on the values of $I_{r1}, \ldots, I_{rM}$. Given the current input vector $V(val_1, \ldots, val_N)$, the basic idea of local search is to use V as the starting point and conduct a local exploration around V. During this process, for every related variable I_{rk}, it replaces the val_{rk} in V with its neighborhood values, denoted as $val_{rk}+\Delta$, while keeping all the other variable values fixed (line 5-13). The newly generated neighboring vectors are

then evaluated against the goal of reaching $brch_{tgt}$. Therefore, an effective fitness function is essential to evaluate the quality of the current vector and further guide the searching process such that the neighborhood exploration is moving towards the direction of the aim. We choose *branch distance* as the fitness function. When the $brch_{tgt}$ is missed, the *branch distance* is a function that measures how far the execution flow is away from the target route $brch_{tgt}$ (line 6), which makes it a proper estimation of how valuable the current input vector is. In other words, the goal of reaching $brch_{tgt}$ can be transformed to trying to decrease the branch distance down to zero and eventually switch the direction of path. The branch distance of the S_{ITE} can be calculated based on the logical expression of its condition and its evaluation results when running the program with V. For example, given an ITE conditional statement "***if*** ($x<y$) …", the branch distance can be computed as the absolute difference between x and y, i.e., abs(x-y). The branch distance fitness computation for various expressions is listed in Table 7.1.

Table 7.1 Branch Distance Fitness Computation

Expression	**Branch Distance (*brchdist*)**
$x<y$, $x\leq y$ $x>y$, $x\geq y$, $x!=y$, $x=y$	abs(x-y)
bool_var	1
exp_1 && exp_2	*if* (exp_1 && exp_2) min(*brchdist*(exp_1), *brchdist*(exp_2)) *if* (exp_1 && !exp_2) *brchdist*(exp_2) *if* (!exp_1 && exp_2) *brchdist*(exp_1) *if* (!exp_1 && !exp_2) *brchdist*(exp_1) + *brchdist*(exp_2)
exp_1 || exp_2	*if* (exp_1 && exp_2) *brchdist*(exp_1) + *brchdist*(exp_2) *if* (exp_1 && !exp_2) *brchdist*(exp_1) *if* (!exp_1 && exp_2) *brchdist*(exp_2) *if* (!exp_1 && !exp_2) min(*brchdist*(exp_1), *brchdist*(exp_2))
!(exp)	*brchdist*(exp)

As a fitness function, the *branch distance* is also responsible for guiding the search process, in this case, selecting the change value of Δ during the local exploration around the $V(val_1, \ldots, val_N)$. In order to establish the relationship between the two factors, we

conduct a one-pass preprocessing step to acquire knowledge on the impact that changing the value of a given related variable I_{rk} will have on the changing direction of branch distance evaluation of S_{ITE}. During this preprocessing step, we feed the program with a set of randomly generated vectors, with the value of I_{rk} sampling different values from each of its sub-domains, while the values of all the other inputs fixed, the corresponding changes in the branch distance are observed. As a result, a probabilistic model between the two factors is constructed as shown in Figure 7.4, where the $incprob_{inc}$, $incprob_{dec}$ and $incprob_{same}$ indicate the probabilities for the branch distance of S_{ITE} to increase, decrease and keep the same, respectively, by increasing the value of I_{rk}. A similar model can be obtained for value decreasing of I_{rk}, where the corresponding probability set is defined as follows: $decprob_{dec} = incprob_{inc}$, $decprob_{inc} = incprob_{dec}$, $decprob_{same} = incprob_{same}$.

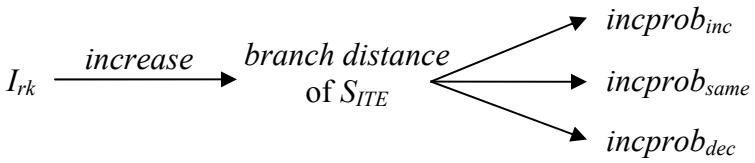


Figure 7.4 Probabilistic Model Between Inputs and Branch Distance

With the probabilistic model and current branch distance evaluation, we are able to first determine the sign (positive or negative) of the Δ stochastically by trying to decrease the fitness. In addition, the weight factors along all the sub-domains of I_{rk} in Figure 7.2 will be adjusted accordingly based on the branch distance value: all the weight factors along the sub-domains greater than val_{rk} are increased by W_{Δ}*$incprob_{dec}$, all the weight factors along the sub-domains lower than val_{rk} are increased by W_{Δ}*$incprob_{inc}$ (line 7), where W_{Δ} indicates a pre-defined weight change. As a result, the magnitude of Δ will be determined according to the weight factor distribution along the sub-domains in its sign direction (line 8). With the updated vector V, the program is executed to see whether the $brch_{tgt}$ can be reached, if yes, the execution profile is added as new transactions to the mining database, otherwise, the data is dropped, we repeat the above process until either the $brch_{tgt}$ is reached or the iteration exceeds certain pre-determined threshold (line 9-12).

Other than related input variables, conditional expressions may also involve constants, which usually define certain comparison boundaries that may sometimes lead to a quick

switching of taken branches, if a local search is conducted around the boundaries. Therefore, we perform a similar local search for variable I_{rk} with respect to all the constants that appear related to I_{rk} in the conditional expression, with val_{rk} being set to be equal to the $const_{rk}$ or $const_{rk}+\Delta$, respectively (line 14-16). The above local search is repeated with a new input vector generated based on the updated weight distribution until either the visiting frequency of the biased branches are more balanced or the iteration exceeds some threshold for the consideration of efficiency. The key of local search based input generation algorithm, combined with probabilistic input models is to utilize the local feature of conditional statements and the statistical execution information. Although it is a heuristic, it can help increase the balance of branch coverage compared to random input generation to a great extent. As a result, it expands the possibility of extracting otherwise unknown CPIs due to low frequency of occurrence, which may consequently enhance the applicability of the path-level partitioned CRA.

7.2.2 Path-Level Partitioning with CRA

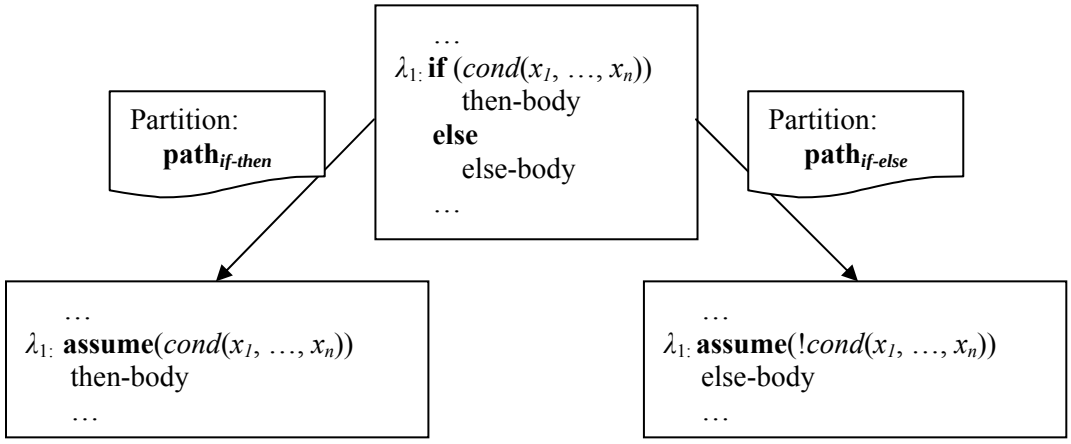


Figure 7.5 An Example of Path-Level Partitioning Directed by PCPI

The proposed path-level partitioning with CRA algorithm consists of two steps. The first step is to construct partitions on the program control flow based on the learned invariants. A mined CPI P_{CPI} is in the form of $\{<\wp::\lambda_1> \ \& \ cond(x_1, \ldots, x_n)\} \Rightarrow \{<\wp::\lambda_2> \ \& \ (y_1 \blacklozenge y_2)\}$. Take the ITE statement as an example, suppose the $cond(x_1, \ldots, x_n)$ is the same as the conditional expression of an ITE statement located at line λ_1(or right after λ_1), the P_{CPI}

partitions the program control flow into two disjoint paths: $path_{if\text{-}then}$ and $path_{if\text{-}else}$, as shown in Figure 7.5. For each path, we create a reduced program by removing the other untaken branch of the given ITE statement and then adding an *assume*() statement correspondingly to constrain the state space with conditions of the taken branch.

The second step is to apply abstraction on each disjoint partition to generate an over-approximate model. For each partition, say $path_{if\text{-}then}$, we perform CRA over the consequent part of the $P_{CPI:} <\wp ::\lambda_2> \& (y_1 \blacklozenge y_2)\}$ using the same method described in Section 6.2.2. Similar abstraction technique is employed on the other partition $path_{if\text{-}else}$, provided there exists another CPI whose condition part is $!cond(x_1, ..., x_n)$.

In particular, we divide the mined CPIs into two categories depending on the code locations where the invariants are discovered:

1) *Localized global*: where the code segment within which λ_2 is located will be executed only when the condition $cond(x_1, ..., x_n)$ at line λ_1 is taken, e.g., the inv_2 in the example in Figure 7.1;

2) *Pure local*: where the code segment within which λ_2 is located will be executed only when the condition $cond(x_1, ..., x_n)$ at line λ_1 is taken, e.g., the inv_1 in the example in Figure 7.1;

For partitioned abstraction, both types of CPIs can be utilized, while for SLP directed CRA, only the localized global CPI can be applied. Note that besides CPIs, for each partition, constraint-reduction-based abstraction over *single* mined invariants SLPs may also be conducted as long as the positions where the SLPs are located are still included in the partitioned code.

The above partitioning algorithm can be easily extended into cases of multiple CPIs. We define two CPIs to be *distinct* from each other if the antecedent of each CPI refers to a different control statement. If multiple distinct CPIs exist in the mined invariant set, the most straightforward way of partitioning can be performed by enumerating all the possible combinations of their conditions. For example, let CPI_1: $\{<\wp ::\lambda_1>\&cond_1 \Rightarrow <\wp ::\lambda_3>\&prop_1\}$ and CPI_2: $\{<\wp ::\lambda_2>\&cond_2 \Rightarrow <\wp ::\lambda_4>\&prop_2\}$ be

the two learned distinct invariants, they will partition the program state space into four sub-spaces constrained by: $cond_1@\lambda_1 \wedge cond_2@\lambda_2$, $\neg cond_1@\lambda_1 \wedge cond_2@\lambda_2$, $cond_1@\lambda_1 \wedge \neg cond_2@\lambda_2$, $\neg cond_1@\lambda_1 \wedge \neg cond_2@\lambda_2$. In the subsequent section, we'll discuss more partition strategies.

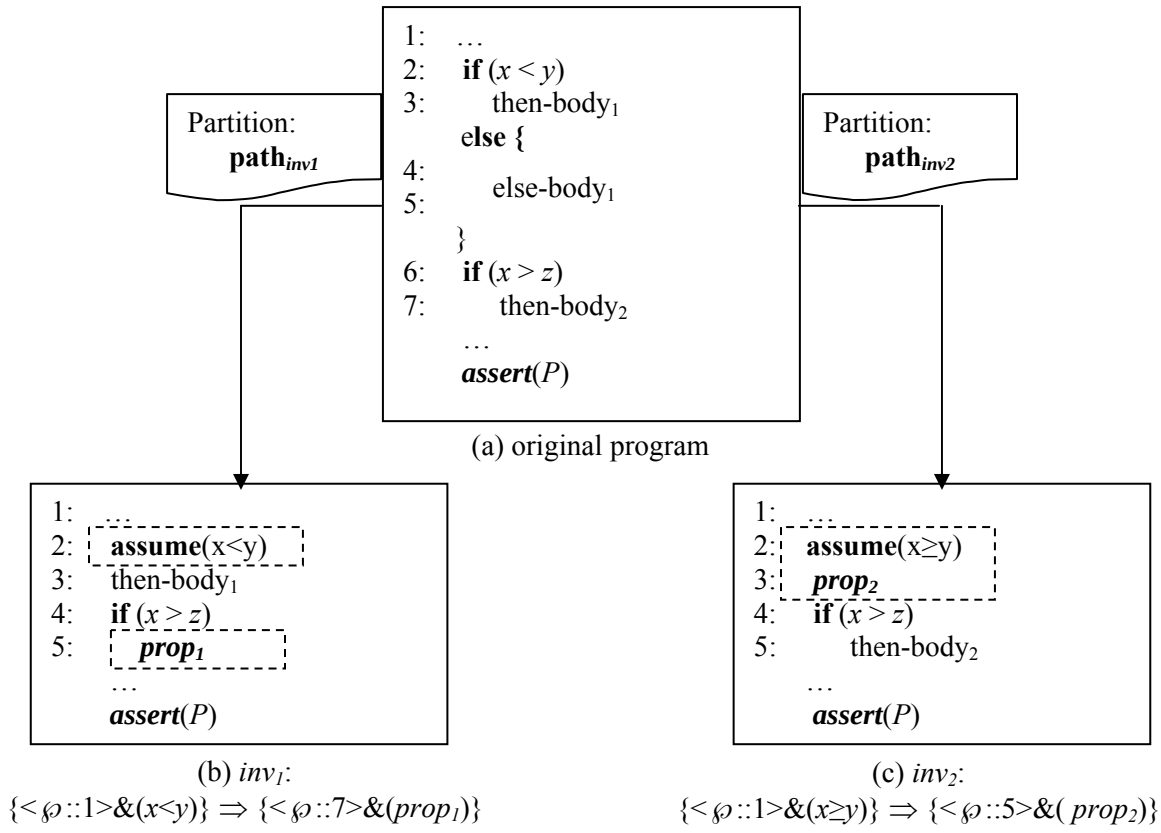


Figure 7.6 An Example for Path-Level Partitioning with CRA

Figure 7.6 shows the results of applying path-level partitioning with CRA on the same example shown in Figure 7.1. The invariants inv_1 and inv_2 are not distinct CPIs, and they partition the code into two paths, since the condition parts of the two invariants are complementary conditional expressions of the same ITE statement. For each partition, CRA is applied using the implied property of each CPI invariant. Here we assume that the then-body$_2$ and else-body$_1$ can be replaced by $prop_1$ and $prop_2$, respectively, according to the rules of invariant-directed CRA.

7.2.3 Invariant Selection and Partitioning Strategy

There are two aspects that may impact the effectiveness of the above partitioned abstraction algorithm: 1) Invariant selection: similar to single-invariant directed CRA abstraction, partitioned abstraction also need to trade the extent of the program complexity reduction brought by abstraction on more invariants with the precision of the abstraction, only valuable invariants in terms of the target property checking will be selected for partitioning and abstraction, and 2) Partitioning strategy: given N selected distinct CPIs, the total number of partition could reach 2^N by enumerating all the possible condition combinations. The number of combinations grows exponentially when N becomes large. If there are too many partitions, the accumulative model checking time might negate the potential performance saving, hence, how to partition the program control flow becomes crucial.

Algorithm *Control-Flow-Partition* **(*CPISet, path_sel_threshold*)**

```
1:   partition_set = composite_partition = Φ
2:   pathSet  = enumeratePath(CPISet)
3:   for each path in the pathSet
4:       if  (every condition expression along the path is a CPI_cond in CPISet)
5:           partition_set= partition_set ∪ path
6:       else  if (exeFreq(path) > path_sel_threshold)
7:               partition_set = partition_set ∪ path
8:           else
9:               composite_partition = composite_partition ∪ path
10:  end for
11:  partition_set = partition_set ∪ composite_partition
12:  return  partition_set
```

Figure 7.7 Partitioning Strategy for the Selected CPI Set

For the invariant selection, since the condition part of the CPI is mainly used for partitioning, we will evaluate the value of a CPI largely based on the implication property

used for CRA abstraction, i.e., the consequent part of the CPI. Let CPI_{cond} and CPI_{impl} denote the condition and the consequent part of CPI, respectively. This comes down to considering the similar combination of three factors described in Section 6.2.3 for SLP selection: 1) *statement distance* from the CPI_{impl} to the target property to be checked; 2) *variable distance*: the overall level distances between every pair of the variables involved in the CPI_{impl} and the property. The closer the two distances, the higher the correlation between the CPI and the verification of the target property might be; 3) *operation complexity*, which estimates the total computational complexity of all the operations involved in the corresponding constraints that would be removed if the CPI is selected, including the operations in the not-taken branch according to CPI_{cond} and the constraints removed for CRA over CPI_{impl}. The higher the complexity estimation, the higher speed-up in model checking might be expected. The overall value of a mined CPI is evaluated by considering the above three factors in an integrated fashion, with the last two factors to be more critical.

For control flow partitioning, we employ a heuristic strategy as shown in Figure 7.7 to reduce the number of partitions. Let *CPISet* be the set of selected CPIs which contains N distinct invariants. We first enumerate all the possible paths based on *CPISet* and obtained a *pathSet* with size 2^N (line 2). Then for every *path* in the *pathSet*, we check whether it can be retained as a partition or merged with other partitions to form a composite partition according to the extent to which the CRA can be applied and the potential size of the state space represented by the *path*(line 3-10). If every related *taken* condition expression along the *path* is caused by the CPI_{cond} of some CPI in the *CPISet*, the path is kept as a partition since such path can take the most advantage of CRA abstraction. Take the two-CPI-set CPI_1: $\{<\wp::\lambda_1>\&cond_1\Rightarrow<\wp::\lambda_3>\&prop_1\}$ and CPI_2: $\{<\wp::\lambda_2>\&cond_2\Rightarrow<\wp::\lambda_4>\&prop_2\}$ in Section 7.2.2 as an example, the path along the $cond_1@\lambda_1 \wedge cond_2@\lambda_2$, will be retained as a partition. Otherwise, we examine the execution frequency of the *path* according to the profile data used for invariant mining, which can be viewed as a rough measurement of the size of the state space represented by the *path*. When the execution frequency is too low, it potentially implies that either too few inputs would fall in the state space constraints posed by the given *path* or the *path* is

infeasible. In such cases, we merge the small partition into *composite_partition* to reduce the unnecessary overhead of model checking caused by repetitive state space searching. Or if the execution frequency of the *path* is greater than some threshold, the *path* is also treated as a separate partition. Therefore, in the above two-CPI-set example, the rest three partitions other than $cond_1@\lambda_1 \wedge cond_2@\lambda_2$ will be determined either to be merged or not based on execution results. Finally, all the merged *composite_partition* is treated as an individual partition (line 11).

7.3 CEGAR with Partitioned CRA

Algorithm *CEGAR-Partitioned-CRA*(**M_{orig}**, ***prop***, ***partition_set***)

```
1:   for every partition in the partition_set
2:       M_abs, partition = Build_CRA(CPIs(partition))
3:       Res = Model_Checking (M_abs, partition, prop)
4:       if (Res = SAT)
5:           if (M_abs, partition != M_orig, partition and Res.cex is spurious)
6:               M_abs, partition' = Cex_Guided_Refinement(Res.cex, M_abs, partition)
7:               M_abs, partition = M_abs, partition'
8:               goto line 3
9:           else return cex          // property prop is falsified in the partition
10:  end for
11:  return verified
```

Figure 7.8 CEGAR with Partitioned CRA

Built on the partitioned abstraction technique, we propose a new CEGAR framework for software bounded model checking. Given the original program M_{orig} and the target property to be checked *prop*, let *partition_set* be the set of partitions obtained using the control flow partitioning strategy in Section 7.2.3, for every *partition* in the *partition_set*, we conduct model checking using algorithm CEGAR_CRA in Section 6.3.1(line 2-9). The abstraction of each partition, constructed based on the corresponding consequent part of the CPIs from where the *partition* is constructed, is refined iteratively until either the

prop holds in the abstract model of the *partition* or a true counterexample *cex* is found within the *partition*. For the latter case, the entire algorithm stops from checking the rest partitions and returns with the *cex*. Otherwise, the work flow goes to the next partition, and repeats all the above process. If no *cex* is found after iterating all the partitions in the *partition_set*, then we can claim that the *prop* is satisfied in M_{orig}. From this algorithm, we can see that, the partitioned CRA not only facilitates the property verification through over-approximation based abstraction to the *partition* with reduced complexity, but also localizes the property falsification within an under-approximation based *partition*.

7.4 Experimental Results

We conducted bounded model checking experiments on several C programs, including both the academic algorithms as well as real-world software applications (various modules from a software defined radio system PSCR [72]). For benchmarks that contain loops, loop unrolling is conducted before collecting profile data up to a pre-defined depth (no smaller than the unwinding depth used for BMC), since path partitioning cannot be directly conducted within the loop. Then, we apply association rule mining for each benchmark to discover potential invariants over the profiled data, which is collected via dynamic program execution either for N=100 times on selected inputs generated for balanced branch coverage or when the transaction count in the mining database exceeds 5000, whichever comes first. Both potential invariants in the form of SLP and CPI (s_{th}=0.4, c_{th}=1.0) were mined and validated as a one-pass process before selected for partition and abstraction. For assertion verification, we used CBMC (C Bounded Model Checking) [6] as the base model checker.

Table 7.2 gives the model checking results of partitioned CRA and compares them to the original BMC as well as BMC with invariant-directed CRA approach proposed in Chapter 6 (The latter is denoted as SLP-CRA). For abstraction refinement, the runtime of each iteration is listed in a parenthesis and we use '**;**' to separate the results of different functionally disjoint partitions. In this chapter, we focus primarily on properties that hold when the model checker returns UNSAT, while the case of property falsification is also covered, like the last benchmark *run_classify* (SAT)). For each benchmark, results of

different unwinding depth or size of major data structure, e.g., arrays, are provided to demonstrate the effectiveness of our approach in handling various sizes of benchmarks and the trend of performance savings. Moreover, in order to have a fair comparison, we performed SLP-CRA abstraction using the same SLPs as well as the localized global CPIs that can be applied globally as in the partitioned abstraction.

Table 7.2 Bounded Model Checking Results with Partitioned CRA

Benchmark	Depth/Size	Orig. BMC	SLP-CRA (Chapter 6)	Partitioned CRA	
				Partitions	Total
BinarySrch	14	1141	(12, 1297)	(24, 913) ; (15, 26)	**978**
	15	4209	**(4,1027)**	(131, 589) ; (15, 28)	**763**
	16	8479	**(81, 2406)**	(62, 1895) ; (12, 79)	**2048**
PolicyEng	20	854	**(7, 712)**	(15,51) ; (7,26) ; (3,221) ; 317	**640**
mod_cvsd_decode	4	2745	**363**	413 ; 431	**744**
	5	1819	**766**	916 ; 482	**1398**
	6	3033	**1389**	1038 ; 992	**2030**
mod_cvsd_encode	3	905	**359**	293 ; 152	**445**
	5	3175	**910**	751 ; 854	**1605**
train_classify	12	1256	**1005**	35 ; 1	**36**
	15	TO	TO	237 ; 1	**238**
run_classify	15	1880	2017	1 ; 1646	**1647**
	18	6372	**5529**	1 ; 4857	**4858**
	20	TO	**7571**	1 ; 7871	**7872**
prog1	12	996	**856**	(5,486)	**491**
	15	6158	**2898**	(10, 2054)	**2064**
	18	TO	**4025**	(15, 4038)	**4053**
prog2	12	576	641	(1, 530)	**531**
	15	OM	OM	(2, 1403)	**1405**
	18	OM	OM	(2, 4732)	**4734**
prog3	10	1244	2034	(1,971)	**972**

	12	2702	TO	(1,1717)	**1718**
	15	TO	**1135**	(2,3648)	**3650**
run_classify(SAT)	20	763	**189**	170	**170**
	25	5853	**622**	403	**403**

(in sec, TO=10,000 sec, OM=Out of Memory)

From the results in Table 7.2 we can observe that, both SLP-CRA and partitioned abstraction can achieve significant performance improvement(denoted as bold) over directly applying model checking on the original programs in nearly all benchmarks. In particular, the partitioned abstraction outperformed SLP-CRA in the number of cases as well as the average extent of performance savings compared to the original BMC. Up to 40 times speedup can be achieved. We used underlined red to represent such cases when the partitioned abstraction runs better than SLP-CRA. For example, for benchmark "*train_classify*" under the unwinding depth of 15, both the original BMC and SLP-CRA abstraction timed out at a limit of 10,000 seconds while the partitioned CRA divides the original program into two paths, and only a total of 238 (237+1) seconds were needed to verify the property in both partitions. These results further validate the effectiveness of partitioned CRA in expanding the strength and utilization of invariant-directed abstraction, and the accumulative performance of a set of separate abstractions over each individual reduced complexity partition tends to demonstrate a promisingly better outcome than the abstraction over the entire program. In addition, our branch coverage directed input generation algorithm also demonstrated the power of broadening the scope of invariant mining. For example, benchmarks prog1, prog2 and prog3 contain very biased branches such that only CPIs for the biased paths would be mined if the underlying mining database is constructed from random program execution. This will greatly limit the applicability of the subsequent CRA abstraction. With our local-search-based input generation algorithm, we were also able to discover more CPIs for less frequently visited paths such that more partitions can be involved in the abstraction, hence, drastically improve the performance of assertion verification.

However, we also found that in some situations, SLP-CRA can work better than partitioned CRA, like benchmarks *mod_cvsd_decode* and *mod_cvsd_encode*. The

possible reason behind this may due to the complexity/criticality of the shared code portions among different partitions with respect to the target property verification, since these parts will be reasoned repeatedly in every partition, but only once in SLP-CRA approach. The overall runtime of the partitioned abstraction might get hurt by such repetitive overhead if the complexity of shared parts is significant and the number of partitions is high, especially when the mined CPIs used for abstraction are mostly localized global. Therefore, it is necessary to make appropriate evaluation of the potential savings and the overhead so as to make the best out of the CRA abstraction.

The last benchmark demonstrates the power of partitioned CRA in property falsification. A true counterexample was obtained in one of the partitions, with up to >10 times of speedup compared to the original case, and it also worked slightly better than SLP-CRA since the model checking was conducted on the reduced program rather than the entire code, given both were applied with the same over-approximated abstraction.

7.5 Conclusions

In this chapter, we further extended the invariant-directed CRA abstraction technique presented in Chapter 6 and combined it with control flow partitioning scheme under the guidance of CPI invariant mining. The partitioned abstraction boosts the applicability of CRA technique by localizing CRA and refinement within each individual partition, such that local invariants can also be involved in abstraction. Moreover, rather than applying CRA abstraction on the entire program, partitioned abstraction conducts abstraction over program portions with already reduced complexity, which enables more aggressive performance gains in both property verification and falsification. We further applied local-search-based input generation algorithm with the goal of balancing branch coverage, such that the mining based on selected input execution could discover more useful CPIs to fully promote the effect of the partitioned CRA. Experimental results showed that model checking with path-level partitioned CRA can achieve significant performance improvement over the original bounded model checking, with up to 40 times of speedup, as well as over the invariant-directed CRA technique in most cases.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this dissertation, we explored various hybrid techniques to tackle the intricate problem of software verification, especially for software model checking. The goal of software verification is to ensure that the designed program satisfies the required specification, generally through either dynamic testing or static formal verification techniques, which suffer the problems of incompleteness and state space explosion, respectively. In our work, we leveraged the lightweight dynamic techniques, usually associated with program execution, to extract useful information out of the code. By utilizing such information in solving the problems of software verification, enhanced effectiveness and scalability can be achieved via the synergies between dynamic and static techniques.

For software testing, we presented new Internal Variable Range (IVR) coverage metric under the guidance of dynamic program execution data. By running the program with random inputs, we project the program input domains onto approximate value ranges of selected critical internal variables, and then partition the internal variable domain into sub-ranges based on the data value distribution as well as the program structure information related to the internal variables. With this new metric, we are able to provide an effective estimation of the fault detection capability of an input test set, especially for programs that are arithmetic computation intensive. Besides, a sub-range transition

system was also proposed to facilitate the internal variable range coverage oriented automatic test data generation process. Experimental results demonstrated the effectiveness of this model in generating test sets with higher IVR coverage.

For software formal verification, we took advantage of the less expensive dynamic techniques to alleviate the burden of formal reasoning. Hence, the complementary benefits of both static and dynamic techniques can be fully utilized, while compensating the weaknesses from each. In addition, when combined with abstraction mechanisms, which is considered to be one of the most powerful methods for reducing the system complexity, the interactions among hybrid analyses demonstrate much more promising strengths on improving the scalability of software model checking. In this work, we explored the hybrid of dynamic techniques with three types of abstractions: exact, under-approximation and over-approximation.

Firstly, we applied data mining to discover valuable but implicit knowledge regarding code structures, especially the invariants that reveal the hidden key relations among the original high-level program variables, since how effectively these invariants can be obtained during the reasoning process is usually the crucial factor that impacts the performance of property checking. A dynamic-program-execution-directed invariant mining framework was proposed that utilized an association rule mining algorithm to extract potential code-level property invariants based on the concrete execution profile data of the software programs. When these learned invariants were verified and added as constraints to the model checking instances of the software, they helped to significantly reduce the search space by saving the reasoning efforts at the bit-level for identifying the implied relations, thus improves the performance of model checking.

Furthermore, we used the mined invariants to formulate the over-approximation-based program abstraction via relaxing the partial code segment that infers the invariants. We took one step beyond the conventional constraint-reduction-based abstraction by further confining the state space with the learned invariants to retain the essential correlations among the related variables with respect to the property checking. The added invariants improved the precision of abstraction such that less refinement iterations were needed

before the abstraction to contain enough details to reach a conclusion. In addition, when global invariants are not available or sufficiently effective for abstraction and property checking, we proposed to apply path-level partitioning scheme to localize the constraint-reduction-based abstraction over local invariants inferred only in the partition. This approach further expanded the applicability and power of invariant-directed CRA abstraction and promoted the flexibility of applying various abstractions over different partitions. In order to fully utilize the benefits of the partitioned CRA, we also strengthened the extent of CPI mining with branch coverage directed input generation for mining database construction. Model checking frameworks based on the Counter-Example Guided Abstraction Refinement (CEGAR) paradigm associated with the above mined-invariant-directed abstraction techniques were found strongly effective in improving the performance of software model checking, especially for verifying properties that hold.

Last, we integrated the Ant Colony Optimization (ACO) algorithm with under-approximation based abstraction technique to facilitate the construction of path-level code abstraction for software model checking, focusing mainly on property falsification during the early design stage. Artificial ant agents were sent out to explore the concrete state spaces such that the program execution flows would be led towards traversing the promising state space that might refute the property. Centralized daemon actions were also conducted periodically in order to prevent the ant solutions from being trapped in some local optimum. At the end, we identified quality program traces through the pheromone trails laid down by ants during the travelling, and accordingly constructed program under-approximation by biasing the control-flow via ignoring less useful statements. An abstraction-based software model checking framework was then built to search for verification solutions over the reduced program, which was iteratively refined until it contained enough details to either find a real counterexample or prove the property.

We conducted extensive experiments on both academic algorithms and real-world programs to evaluate the effectiveness of the above hybrid techniques. The experimental results further provided strong supports for the following findings:

I. Although the lightweight dynamic techniques associated with dynamic program execution are considered to be incomplete in terms of software state space exploration, they are still able to reveal valuable potential properties regarding program control flow structures, variable relations, variable/expression value ranges, etc., even running with randomly generated input vectors;

II. Compared to the conventional static techniques used for invariant discovering, dynamic techniques are more straightforward and flexible;

III. Invariants play critical roles in property checking, however, the effect of an invariant may be influenced by its location, variables involved, frequency of use, etc.;

IV. The hybrid between the high-level dynamic technique and the bit-level static reasoning clearly exhibits strong power in improving the performance of software verification, especially in combination with various abstraction as well as partition techniques, which brings promising direction towards enhancing the scalability of software model checking.

8.2 Suggestions for Future Work

The methods and experimental progress presented in this dissertation also open up new research directions that deserve future investigations to improve the current work. We will suggest some of the aspects in more detail in the following subsections.

8.2.1 Partitioning with Non-Uniform Abstraction Techniques

In Chapter 7, we investigated integrating the path-level partitioning scheme with invariant-directed CRA to further enhance the performance of assertion checking by localizing the abstraction. Currently, within every partition, a uniform abstraction technique, i.e., CRA, is employed. Observations showed that, even within the same program, different partitions may vary significantly in complexity, size, control flow and data structure, the uniform abstraction may not be optimal for all the partitions. According to the feature of current partitioning scheme, where every partition is treated

independently, it would be a natural extension to introduce the non-uniform abstractions over different partitions based on the specific characteristic of the code in the partition. For example, predicate abstraction in one partition and CRA abstraction in another.

Compared to the existing mechanism, partitioning with non-uniform abstraction (PNUA) may present higher flexibility and strength in software model checking. For instance, some partition may not have suitable invariants that can be used for abstraction. Hence, it can bring none or only limited savings over the original partition with CRA abstraction. For example, the partition $\neg cond_1@\lambda_1 \wedge \neg cond_2@\lambda_2$ in the two-CPI-set example in Section 7.2.2 does not match any of the condition parts of the two CPIs with the conditional branches taken along its path, thus, no CRA abstraction can be applied. However, with PNUA, we may apply a different abstraction technique other than CRA, e.g., predicate abstraction to the partition, so as to seek for better performance improvement. In addition, PNUA enables the break of limitation on applying just over-approximation based abstraction, which boosts the power of the current BMC framework in property falsification. Under-approximation can also be employed over a partition with already reduced state space to quickly identify the counterexample, especially when the code is buggy at the early design stage.

8.2.2 Program Structure Directed Invariant Mining for Abstraction

In this dissertation, we conducted invariant mining by simply transforming the original program into a set of indexed profiling statements without considering how these statements are structured in the original code. Therefore, the resulting mining database is composed of kind of isolated transaction data which can only be connected with each other using line numbers. Although purely from the perspective of invariant mining, current strategy can fulfill the task with correct mining results, considering the learned invariants are eventually used for program abstraction, and later on model checking, there still exists a gap between appropriate invariant mining and the effective utilization of the mined invariants.

As we have shown in Chapter 7, in order to maximize the strength of partitioned CRA, it is necessary to apply selected branch-balancing input vectors to run the program so as to

expand the range of CPI mining over previously less frequently visited branches. For invariant mining used for exact abstraction (Chapter 4) and global invariant directed CRA (Chapter 6), which has less strict format regulations than in the partitioned CRAs, it would also be valuable to investigate how various underlying mining database constructed from different input vectors may influence the subsequent program abstraction and the performance of model checking and how can we direct input generation process towards the desired goal under the situations when the target assertion is either known or unknown.

One likely research direction could be the program structure directed input generation. Program structure analysis helps provide useful information on program complexity estimation, data dependency, the relationship between certain code structure and the target assertion, if known, etc. If such information is integrated with the input selection process during profile data collection, we would be able to set up potential structural targets for the input vector generation. For example, it may be more valuable to discover properties right after the execution of some computation intensive statements, since abstraction over these statements can usually help reduce the program complexity significantly. Thus, it is desirable to cover execution traces of these statements frequently enough in the mining database such that they can be eligible for association rule mining. In addition to the program complexity, data dependency also helps identify critical variables in the case when the target assertion is known or eliminates unnecessary redundant mining over unchanged variables. Once we locate the target program structure, the goal-oriented input generation algorithm can then be applied to generate valid inputs. As a result, invariants with higher quality may be learned in terms of program abstraction and assertion verification.

8.2.3 Reusing the Reasoning Results

According to the model checking framework proposed in Chapter 5–7, either the design under verification is a refined abstract model via restoring the removed statements back, or another path-level partition, we performed a new round of bounded model checking on the entire design all over again, starting from the high-level code to CNF transformation,

with no information about the reasoning process of the previous iteration of model checking. By doing this way, we reduce the complexity of applying model checking. However, it might also introduce unnecessary overhead due to the repetitive reasoning.

Intuitively, further work can be directed towards how to reuse the reasoning results in previous iterations of abstraction or partition. Since models between adjacent iterations abstraction refinement share a majority portions of code other than slight difference brought by the restored statements, similarly do the models between partitions, it would be valuable to investigate the bit-level proofing process for the shared portions of constraints, so as to further reduce the reasoning burden of the model checking in the subsequent iterations. Here is a simple example: let A, B, C represent CNF formulas, given a UNSAT model A∩B, representing one partition of the program, using the resolution-based proof analysis techniques for SAT solvers [98, 99], if the *unsatisfiable core* contains only clauses in A, then the next partition, model A∩C is definitely UNSAT, no further model checking is needed.

Besides the above described suggestions for future work, other directions like integration with static analysis techniques, heuristic strategies for invariant selection, etc. can also be further explored to improve the current work. In summary, the hybrid methods for software verification and experimental results presented in this dissertation have laid a solid foundation for continued research.

Bibliography

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs", in *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*(*TACAS*), LNCS. Springer-Verlag, 1999.

[2] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a SAT solver" in *Proceedings of 3rd International Conference on Formal Methods in Computer-Aided Design*(*CAD*), pp. 108–125, 2000.

[3] M. K. Ganai and A. Gupta, "Accelerating high-level bounded model checking" in *International Conference on Computer-Aided Design*(*CAD*), Nov. 2006.

[4] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints", in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 238–252, 1977.

[5] W. Wu and M. S. Hsiao, "Mining global constraints for improving bounded sequential equivalence checking" in *Proceedings of the 43rd annual conference on Design automation,* pp. 743–748, 2006.

[6] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs" in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176, 2004.

[7] G. J. Myers. The Art of Software Testing, John Wiley and Sons, New York, 1979.

[8] G. Brat and W. Visser, "Combining static analysis and model checking for software analysis" in *Proceedings of the 16th Annual International Conference on Automated Software Engineering,* pp. 262–269, 2001.

[9] C. Pasareanu and W. Visser, "Verification of Java programs using symbolic execution and invariant generation" in *Proceedings of the 11th* International *SPIN Workshop on Model Checking of Software*, pp. 164–181, Apr. 2004.

[10] R. Agrawal, T. Imielinski and A.N. Swami, "Mining association rules between sets of items in large databases" in *ACM SIGMOD International Conference on Management of Data*, pp. 207–216, May 1993.

[11] R. Agrawal, T. Imielinski and A.N. Swami, "Database mining: A performance perspective" in *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 6, pp. 914–925, 1993.

[12] D. E. Maydan, J. L. Hennessy and M. S. Lam, "Efficient and exact data dependence analysis" in *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 1–14, 1991.

[13] M. S. Hecht, Flow Analysis of Computer Programs. Elsevier North-Holland Inc. 1977.

[14] A. Tiwari, H. Rueß, H. Saïdi and N. Shankar, "A technique for invariant generation" in *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*(*TACAS*), pp: 113–127, 2001.

[15] J. Fischer, R. Jhala and R. Majumdar, "Joining dataflow with predicates" in *Proceedings of 13th International Symposium on Foundations of Software Engineering*, pp. 227–236, 2005.

[16] N. Dodoo, A. Donovan, L. Lin, and M. D. Ernst, "Selecting predicates for implications in program analysis" in Mar. 2002. Draft. http://pag.csail.mit.edu/~mernst/pubs/invariants-implications.ps.

[17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz and C. Xiao, "The Daikon system for dynamic detection of likely invariants" in *Science of Computer Programming*, 2007.

[18] M. Fuchs, "A data mining approach to support the creation of loop invariants using genetic programming", Tech. Report, TR-ARP-09-98, 1998.

[19] N. Eén, and N. Sörensson, "An extensible SAT-solver", in *Proceedings of 6th International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[20] E. M. Clarke, E. A. Emerson and A. P. Sistla, "Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach" in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'83)*, pp. 117–126, 1983.

[21] E. M. Clarke, O. Grumberg and D. A. Peled. Model Checking. MIT Press, 1999.

[22] J. P. Queille and J. Sifakis, "Specification and verification of concurrent systems in Cesar", in *Proc. of the 5th International Symposium on Programming,* pp. 337–351. Springer-Verlag, 1982.

[23] E. Allen Emerson. *Handbook of theoretical computer science*, chapter Temporal and Modal Logic, pp: 997–1071. Elsevier Science Publishers B.V., 1990.

[24] J. Tian. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*. Wiley-IEEE Computer Society Press, 2005.

[25] E. M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, N.Y., 1981. Springer-Verlag.

[26] E. M. Clarke, D. Kroening, N. Sharygina and K. Yorav, “Predicate abstraction of ANSI-C programs using SAT”, in *Formal Methods in System Design*(FMSD), vol. 25(2-3), pp. 105–127, 2004.

[27] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[28] O. Coudert and J.C. Madre. “A unified framework for the formal verification of sequential circuits”, in *Proceedings of IEEE International Conference on Computer-Aided Design*, 1990.

[29] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[30] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[31] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, pp. 506-521, 1999.

[32] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of Design Automation Conference*, 2001.

[33] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M. B. Dwyer, editor, *Model Checking Software, 8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pp: 103–122. Springer, May 2001.

[34] T. A. Henzinger, R. Jhala and R. Majumdar. “Lazy abstraction”. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, January 2001.

[35] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement", in *Proceedings of the International Conference on Computer Aided Verification* (CAV), pp. 154–169, 2000.

[36] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular Verification of Software Components in C." *Transactions on Software Engineering (TSE)*, Volume 30, Number 6, pp. 388–402, June 2004.

[37] F. Ivančić, Z. Yang, M. K. Ganai, I. Shlyakhter, A. Gupta and P. Ashar. "F-Soft: Software verification platform". In *Proceedings of the International Conference on Computer Aided Verification* (CAV), pp. 301–402, July 2005.

[38] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8): 870-879, 1990.

[39] B. Korel. Automated test generation for programs with procedures. In *International Symposium of Software Testing and Analysis*, p209-215, San Diego, California, USA, 1996.

[40] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1): 63-86, 1996.

[41] E. Diaz, J. Tuya, R. Blanco. "Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search". In *18th IEEE International Conference on Automated Software Engineering*, pp. 310-313, 2003.

[42] M. Roper. Computer aided software testing using genetic algorithms. In *10th International Software Quality Week*, San Francisco, USA, 1997.

[43] C. Michael, G. McGraw, and M. Schatz. "Generating software test data by evolution", in *IEEE Transactions on Software Engineering*, 27(12):1085-1110, 2001.

[44] J. Wegener, K. Buhr, and H. Pohlheim. "Automatic test data generation for structural testing of embedded software systems by evolutionary testing", in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp:1233-1240, New York, USA, 2002.

[45] P. Mcminn, M. Harman, D. Binkley and P. Tonella. "The species per path approach to search-based test data generation", in *International Symposium on Software Testing and Analysis*, pp. 13-23, July, 2006.

[46] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[47] T. Ball. A theory of predicate-complete test coverage and generation. *Microsoft Research Technical Report MSR-TR-2004-28*, 2004.

[48] A. M. Memon, M. L. Soffa and M. E. Pollack. Coverage criteria for GUI testing. In *European Software Engineering Conference (ESEC) and ACM SIGSOFT Foundations of Software Engineering Conference*, Vienna, Austria, Sept. 2001.

[49] N. Kosmatov, B. Legeard, F. Peureux and M. Utting. Boundary coverage criteria for test generation from formal models. In *Proceedings of Fifteenth International Symposium on Software Reliability Engineering (ISSRE'04),* Saint-Malo, France, pp: 139-150, 2004.

[50] K. Y. Cai, T. Jing, C. G. Bai. Partition testing with dynamic partitioning. In *29th Annual International Computer Software and Applications Conference (COMPSAC)*, July 2005.

[51] B. Jeng, E, J Weyuker. A simplified domain-testing strategy, *ACM Transactions on Software Engineering and Methodology*, 3(3):254-270, July 1994.

[52] R. A. DeMillo, R. J. Lipton, F. G. Sayward. Hints on test data selection: Help for the practicing programmer, Computer 11(4):34-41, April 1978.

[53] B. Biezer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[54] D. Hamlet. “Partition testing does not inspire confidence”. *IEEE Transactions on Software Engineering,* Vol. 16, No. 12: 1402-1411, Dec. 1990.

[55] G. Beni, J. Wang. Swarm Intelligence in Cellular Robotic Systems, in Proceeding of NATO Advanced Workshop on Robots and Biological Systems, Tuscany, Italy, June 26–30, 1989.

[56] E. Bonabeau, M. Dorigo and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.

[57] M. Dorigo, V. Maniezzo and A. Colorni, “Ant system: optimization by a colony of cooperating agents", *IEEE Transactions on Systems, Man, and Cybernetics–Part B*, 26 (1): 29–41, 1996.

[58] K. Doerner, W. Gutjahr, “Extracting test sequences from a Markov software usage model by ACO”, in *Proceedings of Genetic and Evolutionary Computation Conference*(GECCO), pp. 2465-2476, 2003.

[59] P. McMinn, M. Holcombe, “The state problem for evolutionary testing”, in *Proceedings of Genetic and Evolutionary Computation Conference*(GECCO), pp. 2488-2500, 2003.

[60] H. Li, and C. P. Lam, “An ant colony optimization approach to test sequence generation for state-based software testing”, in *Proceedings of the International Conference on Quality Software*, pp: 255–262, Sept. 2005.

[61] K. Ayari, S. Bouktif and G. Antoniol, “Automatic mutation test input data generation via ant colony”, in *Proceedings of Genetic and Evolutionary Computation Conference*(GECCO), pp: 1074–1091, 2007.

[62] R. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman and B. Brady, “Deciding bit-vector arithmetic with abstraction”, in *Proceedings of Tools and*

Algorithms for Construction and Analysis of System(TACAS), pp: 358-372, March, 2007.

[63] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler and D. L. Dill, "CMC: A pragmatic approach to model checking real code", in *Proceedings of Symposium on Operating Systems Design and Implementation* (OSDI), pp: 75-88, 2002.

[64] B. Gulavani, T. Henzinger, Y. Kannan, A. Nori, S. K. Rajamani, "SYNERGY: A new algorithm for property checking", in *Proceedings of the 14th Annual Symposium on Foundations of Software Engineering*, pp: 117-128, 2006.

[65] G. Yorsh, T. Ball and M. Sagiv, "Testing, abstraction, theorem proving: better together!" in *Proceedings of International Symposium on Software Testing and Analysis* (ISSTA), pp: 145-156, 2006.

[66] C. S. Pasareanu, R. Pelanek and W. Visser, "Concrete search with abstract matching and refinement", in *Proceedings of Computer Aided Verification* (CAV), pp. 52-66, 2005.

[67] G. J. Holzmann and R. Joshi, "Model-driven software verification", in *Proceedings of SPIN*, pp.76-91, 2004.

[68] C. Chou and D. Peled, "Formal verification of a partial-order reduction technique for model checking", In *Proceedings of Tools and Algorithms for Construction and Analysis of System*(TACAS), pp. 241–257, 1996.

[69] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS", in *Proceedings of Computer Aided Verification* (CAV), June 1997.

[70] S. Vasudevan and J. A. Abraham. "Static program transformations for efficient software model checking", in *Proceedings of IFIP Congress Topical Sessions*, pp.257-282, 2004.

[71] N. He and M. S. Hsiao, "Bounded model checking of embedded software in wireless cognitive radio systems", in *Proceedings of IEEE International Conference on Computer Design(ICCD)*, 2007.

[72] T. Rondeau, B. Le, D. Maldonado, D. Scaperoth, and C. W. Bostian, "Cognitive radio formulation and implementation", in *IEEE Proceedings of CROWNCOM*, Mykonos, Greece, 2006.

[73] T. Ball, R. Majumdar, T. Millstein and S. K. Rajamani. "Automatic predicate abstraction of C programs", in *Programming Language Design and Implementation*(PLDI), SIGPLAN Notices 36(5), pp. 203–213, 2001.

[74] C. Wang, H. Kim and A. Gupta. "Hybrid CEGAR*:* combining variable hiding and predicate abstraction", in *Proceedings of International Conference on Computer Aided Design*(ICCAD)*,* 2007.

[75] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, Cambridge, MA, 2004.

[76] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata- Theoretic Approach*. Princeton University Press, 1994.

[77] T. Ball and S.K. Rajamani. *Boolean programs: A model and process for software analysis*. Technical Report 2000-14, Microsoft Research, February 2000.

[78] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. *Counterexample-guided abstraction refinement for symbolic model checking*. Journal of ACM, 50(5):752–794, 2003.

[79] E. Clarke, R. Enders, T. Filkorn and S. Jha, "Exploiting symmetry in temporal logic model checking", in *Formal Method in System Design*, Vol. 9(1/2), pp. 41–76, 1996.

[80] E. Emerson and A. Sistla, "Symmetry and model checking", in *Formal Methods in System Design, Vol. 9(1/2)*, pp. 105–130, 1996.

[81] E. Emerson and R. Trefler, "From asymmetry to full symmetry: New techniques for symmetry reduction in model checking", in *Correct Hardware Design and Verification Methods* (*CHARME*). Lecture Notes in Computer Science, vol. 1703, pp. 142–156, 1999.

[82] P. Godefroid, D. Peled and M. Staskausk, "Using partial order methods in the formal verification of industrial concurrent programs", in *Proceedings of the International Symposium on Software Testing and Analysis*(ISSTA), pp. 261–269, 1996.

[83] D. Peled, "All from one, one from all: on model checking using representatives", in *Proceedings of the 5th International Conference on Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 697. pp. 409–423, 1993.

[84] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction", in *ACM Transaction on Programming Languages and Systems (TOPLAS) 16*, 5 (Sept.), 1512–1542, 1994.

[85] W. Lee, A. Pardo, J. Jang, G. Hachtel and F. Somenzi, "Tearing based abstraction for CTL model checking", in *Proceedings of the International Conference of Computer-Aided Design* (*ICCAD*)*,* pp. 76–81, 1996.

[86] A. Pardo and G. Hachtel. "Incremental CTL model checking using BDD subsetting", in *Design* Automation Conference (DAC), 1998.

[87] L. Zhang, M. R. Prasad, M. S. Hsiao, and T. Sidle. "Dynamic abstraction using SAT-based BMC", in *ACM/IEEE Design Automation Conference* (DAC), pp.754–757, San Jose, CA, June 2005.

[88] N. Amla and K. L. McMillan. "A hybrid of counterexample-based and proofbased abstraction", in *Formal Methods in Computer Aided Design* (FMCAD), pp. 260–274, Nov. 2004.

[89] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. "SAT based abstraction refinement using ILP and machine learning", in *Computer Aided Verification* (CAV), pp. 265–279. Springer-Verlag, July 2002. LNCS 2404.

[90] K. L. McMillan and N. Amla. "Automatic abstraction without counterexamples", in *Tools and Algorithms for Construction and Analysis of Systems* (TACAS), pp. 2–17. Springer, 2003. LNCS 2619.

[91] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. "Iterative abstraction using SAT-based BMC with proof analysis", in *International Conference on Computer-Aided Design*(CAV), pp. 416–423, Nov. 2003.

[92] R. Sebastiani, S. Tonetta and M. Y. Vardi, "Property-driven partitioning for abstraction refinement", in *Tools and Algorithms for Construction and Analysis of Systems*(TACAS), pp. 389–404, 2007.

[93] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. "Abstractions from proofs", in *The 31st Annual ACM SIGPLAN – SIGACT Symposium on Principles of Programming Languages* (*POPL*), pp. 232–244, 2004.

[94] X. Cheng and M. S. Hsiao, "Simulation-based internal variable range coverage metric and test generation model", in *Proceedings of International Conference on Software Engineering and Applications* (SEA), Nov. 2006.

[95] X. Cheng and M. S. Hsiao. "Simulation-directed invariant mining for software verification", in *Proceedings of the IEEE Design Automation and Test in Europe Conference* (DATE), pp. 682–687, Mar. 2008.

[96] X. Cheng and M. S. Hsiao, "Ant colony optimization directed program abstraction for software bounded model checking*",* in *Proceedings of IEEE International Conference on Computer Design(ICCD)*, pp. 46–51, Oct. 2008.

[97] M. Müller-Olm, D. Schmidt and B. Steffen. "Model Checking: A Tutorial Introduction" In *Proceedings of the International Static Analysis Symposium*

(SAS'99), volume 1694 of Lecture Notes in Computer Science, pp. 330-354. Springer-Verlag, Berlin, 1999.

[98] L. Zhang and S. Malik. "Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications", in *Proceedings of the IEEE Design Automation and Test in Europe Conference*(DATE), 2003.

[99] E. Goldberg and Y. Novikov. "Verification of proofs of unsatisfiability for CNF Formulas", in *Proceedings of the IEEE Design Automation and Test in Europe Conference*(DATE), 2003.

[100] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter and C. Wang. "Using statically computed invariants inside the predicate abstraction and refinement loop" in *International Conference on Computer Aided Verification*, pp. 137–151, Springer LNCS 4144, Aug. 2006.

[101] P. McMinn. "Search-based software test data generation: A survey", *Software Testing, Verification and Reliability*, 14(2):105-156, 2004.

[102] J. Clarke. "Automatic test generation from a behavioural model", Software Quality Week Conference, May 1998.

[103] S. Rosaria. "Applying models in your testing process". Information and Software Technology, 42:815–824, 2000.

[104] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. "Efficiently computing static single assignment form and the control dependence graph". ACM Transactions on Programming Languages and Systems, 13(4): 451-490, October 1991.

[105] E. I. White and L. J. Cohen. "A domain strategy for computer program testing", IEEE Transaction on Software Engineering, SE-6(3):247–257, May 1980.

[106] G. Birkhoff, Lattice Theory, 7th printing of the 1967 3rd Edition, American Mathematical Society Colloquium Publications, Vol. 25, American Mathematical Society, Providence, Rhode Island, 1993.

[107] K. N. King and A. J. Offutt. “A Fortran language system for mutation-based software testing”, Software-Practice and Experience, 21(7): 685-718, July 1991.