University of Nebraska - Lincoln

DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses, Dissertations, and Student Research

Computer Science and Engineering, Department of

Fall 12-2012

A Unifying Approach to Behavioral Coverage

Elena Sherman
University of Nebraska-Lincoln, esherman@cse.unl.edu

Follow this and additional works at: https://digitalcommons.unl.edu/computerscidiss

Part of the Computer Engineering Commons

Sherman, Elena, "A Unifying Approach to Behavioral Coverage" (2012). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 50.
https://digitalcommons.unl.edu/computerscidiss/50

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Computer Science and Engineering: Theses, Dissertations, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

A UNIFYING APPROACH TO BEHAVIORAL COVERAGE

by

Elena Sherman

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professor Matthew B. Dwyer

Lincoln, Nebraska

December, 2012

A UNIFYING APPROACH TO BEHAVIORAL COVERAGE

Elena Sherman, Ph. D.

University of Nebraska, 2012

Adviser: Matthew B. Dwyer

Developing methods for validating that a program works as intended is one of the key research areas in software engineering. Ideally a program P must exhibit its expected behavior, or property, ϕ on all of its inputs, i.e., $P \models \phi$. The software engineering community has developed various program analysis approaches to assess whether $P \models \phi$. In general, these approaches can be partitioned into dynamic and static program analysis. The former execute P on a particular input and checks that the execution conforms to ϕ. The latter interprets the code of P and check that on all possible executions of P the property ϕ holds. Unfortunately, in general neither dynamic nor static analysis can independently determine $P \models \phi$.

The idea of combining information computed by different analyses has been circulating in the research community since the mid 1960's and has shown the benefits of analyses unification. Several approaches have been developed for combining multiple static analyses, and combining static and dynamic analyses. These approaches mainly deal with combining the intermediate result of one analysis to help another analysis with deciding $P \models \phi$. This dissertation takes an alternative approach by allowing each analysis to determine $P \models \phi$ under some conditions. Then, combining the final results of such analyses causes $P \models \phi$ to hold under a weaker condition until, ultimately, an unconditional final result is produced.

This dissertation formalizes and implements a unification framework that combines computed information from analyses and disseminates that information among other analyses. This framework is extensible since the only requirement that an analysis should

satisfy is to have querying and reporting capabilities. Conducted in this context of the unification framework, our experiments have shown that combining results from a diverse set of analyses produces weaker conditions for $P \models \phi$ than analyses can achieve operating in isolation.

ACKNOWLEDGMENTS

First, I would like to thank my husband, Glen, and my mother, Maria. Without their support and patience this dissertation would not have been possible.

I also thank my adviser and mentor, Dr. Matthew Dwyer, for the guidance and encouragement he provided and, of course, for keeping his door always open. He puts significant effort in developing his students into research scientists with high ethical values.

Further, I would like to thank the members of my committee: Dr. Sebastian Elbaum, Dr. Myra Cohen and Dr. Christine Kelley for their input and valuable comments. Dr. Elbaum and Dr. Cohen have been an inseparable part of my graduate studies. Working with them inside and outside the classroom environment has greatly influenced my career path and has undoubtedly benefited me as a researcher.

I would also like to thank my colleagues in the ESQuaReD lab and CSE department for being ever available to help and answer questions. I especially would like to thank my colleague and friend, Katie Stolee, who provided needed compassion and moral support all the way through this academic journey.

GRANT INFORMATION

This research was supported in part by National Aeronautics and Space Administration under grant number NNX08AV20A and Air Force Office of Scientific Research under award #FA9550-10-1-0406

Contents

Contents **vi**

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Introductory Background on Program Properties and Analyses 4

1.1.1 Dynamic Analysis 6

1.1.2 Static Analysis 8

1.2 Partial Results of Static Analysis 10

1.3 Conditional Static Analysis 12

1.4 Combining Path Property Coverage 15

1.5 Thesis Statement and Contributions 17

2 Background and Related Work **19**

2.1 Dynamic Program Analysis 21

2.1.1 Testing and Adequacy Criteria 21

2.1.2 Dynamic Symbolic Executions 22

2.2 Static Program Analysis 25

2.2.1 Data-Flow Analysis 25
2.2.2 Predicate Abstraction 28
2.2.3 Combining Program Analysis 30
2.3 Conditional Program Analysis 32
2.3.1 Conditional Data-flow Analysis 35

3 Unification Framework **38**
3.1 Generalized View of Analysis 39
3.2 The Space of Π and L_D parameters 44
3.3 The Unification Framework Overview 46

4 Path Property Coverage **49**
4.1 Properties and Paths 50
4.2 Defining Sets of Paths 52
4.3 Encoding Path Languages 53
4.3.1 BDDs for PPC 55
4.3.2 Iterative Paths 57
4.3.3 Assessing and Querying PPC 58

5 Instantiation of the Unification Framework **63**
5.1 Unification Framework 64
5.1.1 Algorithm 64
5.1.2 Implementation 66
5.2 Adapting Existing Analyses 67
5.2.1 Conditional Data-Flow Analysis 68
5.2.1.1 Algorithm 68
5.2.1.2 Implementation 71

5.2.2 Adapting a Conditional Data-Flow Analysis 72
5.2.2.1 Algorithm 72
5.2.2.2 Implementation 74
5.2.3 Adapting Dynamic Symbolic Execution 74
5.2.3.1 Algorithm 74
5.2.3.2 Implementation 76

6 Evaluation **78**

6.1 Experiment Set Up 79
6.1.1 Analyses 79
6.1.2 Programs 80
6.1.3 Evaluation Infrastructure 80
6.2 RQ1: Comparing CSA and DSE 82
6.2.1 Experiment Design 83
6.2.2 Results and Analyses 84
6.3 RQ2: Comparing the Result of CSA on $\Pi' \subset \Pi$ 86
6.3.1 Experiment Design 86
6.3.2 Results and Analyses 87
6.4 RQ3: Comparing a Single Analysis and a Set of Analyses 89
6.4.1 Experiment Design 91
6.4.2 Results and Analyses 93
6.5 Summary and Limitations of Results 95

7 Conclusion **101**

7.1 Future Work 103

Bibliography **105**

List of Figures

1.1 Partial result example: source code (left), CFG (middle), and the sign analysis result (right) 6

1.2 Partial result example: DSE on inputs $(-1, 0)$ (left), and $(1, 0)$ (right) 11

1.3 Conditional analysis example: source code (left), CFG (middle), and signs analysis (right) 14

1.4 *Conditional* signs analysis (right) and test case with DSE (left) results 16

2.1 Lattices for $zero$ domain (left), $signs$ domain (middle) and $\mathcal{P}(\mathbb{Z})$ domain (right). 29

3.1 Different analysis instantiations based on Π and L. 45

3.2 Concept of combining results of different analyses. 45

3.3 Generating and Exploiting PPC 47

4.1 PPC store BDD encodings for Example 56

4.2 Explanation of subtree coverage criterion 59

4.3 Special case of sub-tree coverage for SA:b_3: execution tree(left), BDD(middle), CFG(right) 60

6.1 RQ3 results for CSA$_{zero}$ and `one_tcas` 97

6.2 RQ3 results for CSA$_{zero}$ and `wbs` 98

6.3 RQ3 results for CSA$_{sign}$ and `one_tcas` 99

6.4 RQ3 results for CSA$_{sign}$ and `wbs` 100

List of Tables

6.1 Artifacts for the experiments 81

6.2 Data for the SA part of the experiment 84

6.3 Data for the DSE part of the experiment 85

6.4 RQ2 results for `asw` 90

6.5 RQ2 results for `one_tcas` 90

6.6 RQ2 result for `wbs` 90

Chapter 1

Introduction

An important problem in software engineering is assessing to what degree a program P works as expected. To address this problem the software engineering community has developed several widely used techniques such as dynamic and static analyses. This dissertation focuses on analysis techniques that analyze sets of program executions, which are referred to as *behaviors*, with the goal of determining certain *facts* about those behaviors, e.g., assertion holds, no null references.

The nature of dynamic analysis is such that it examines a single program behavior at a time and is always able to decide whether the fact for it holds or not. Using this technique one can validate a program by analyzing each of its behaviors. However, for a program with infinitely many behaviors, i.e., programs containing loops checking all behaviors of P by the means of dynamic analysis, is intractable. Thus, in practice, dynamic analysis cannot be used to examine all behaviors of P. On the contrary, static analysis considers all of P's behaviors at once, even if there are infinitely many of them. Unfortunately the set of analyzed behaviors may contain infeasible ones – behaviors that do not appear in actual program executions. Due to its over-approximating nature, static analysis is not able to determine whether the fact fails to hold on feasible or infeasible behaviors. Therefore this type of analysis can potentially produce no definitive answer.

This summary of static analysis presents the customary view of the technique which is to compute the answer for *all* program behaviors. Yet, if a static analysis is able to calculate a definitive answer for only a subset of behaviors then this answer is discarded. For example, for the program on Figure 1.1 a static analysis can determine that behaviors executing the true branch of the second conditional statement will never violate the assertion since this branch is determined to be infeasible. In another program example on, Figure 1.3, the same analysis can produce a definitive result when the set of behaviors analyzed is restricted to those that execute the true branch of the first conditional statement. Utilizing such partial results of static analysis can lead to better assessment of program behaviors by

allowing each analysis to work with the set of behaviors for which it can calculate definitive results. Then partial results from different program analyses can be combined to examine all behaviors of P.

The key challenge in implementing this approach is deriving an encoding of program behaviors that can be used across different analyses, i.e., representation of program behavior should be independent of analysis type. Moreover, abstracting away from analysis implementation detail makes it possible to incorporate dynamic analysis since it can be viewed as an analysis on a single behavior. Additional requirements on the encoding include the ability to compose behaviors and querying the composed behaviors for completeness. An encoding that satisfies these requirements can be used to design a unifying framework that composes the analyzed behaviors from different analyses and determines when all behaviors have been examined.

This dissertation proposes such a unifying framework for program behaviors that makes it possible to salvage partial results of a static analysis, to exploit the power of static analysis by allowing it to examine a subset of behaviors, and to enable the conjunction of the results of program behaviors from different analyses. The goal of the dissertation is to explore how this unification framework affects the evaluation of the conformance of P's behaviors to the expected behaviors – ones that meet explicitly specified correctness properties.

This introductory chapter explains how an expected behavior is characterized, what conformance to the expected behavior means, and the ways to evaluate that conformance. Next, the chapter briefly explains traditional dynamic and static analyses techniques for verifying P's behavior and drawbacks of each analysis technique. These explanations are followed by three research questions exploring methods that benefit from the proposed unification framework. Each question is accompanied by an example that illustrates in detail the benefits of methods. The introduction concludes by posing the thesis statement and highlighting the additional contributions of the dissertation.

1.1 Introductory Background on Program Properties and Analyses

P is said to behave correctly, i.e., conform to its expected behavior, when P satisfies a predetermined set of requirements. For instance, a text editing application is required to not end unexpectedly on a Save operation or a Square Root program must return the square root of its input value. Depending on the complexity of P and the criticality of the requirements, the level of effort that developers invest in assuring P's compliance with properties may vary significantly. For example, if the Square Root program is a component of an avionic navigation system then it is imperative to validate that on all possible inputs this program returns the correct value. However, if the text editing application fails to save a file, then some editing changes are lost, but this may be judged to be noncritical and only worth a small amount of effort to validate.

The validation that P satisfies its requirements is becoming an increasingly automated process. For this reason requirements are translated from their English descriptions to formulae composed of predicates over the observable executable behavior of P, i.e., events or data values. This formal description of a requirement is commonly referred to as a property – denoted ϕ. For example, the obvious requirement, name it ϕ_{SR}, for the Square Root program can be expressed via the output (x_{out}), and input (x_{in}) values as $\phi_{SR} \triangleq \forall x_{in}$: $x_{in} == x_{out} * x_{out}$. Generally a property incorporates two parts: *what* to verify and *where* to verify. The "what" component is commonly described by a predicate, e.g., $x_{in} == x_{out} * x_{out}$, and the "where" component qualifies the predicate, e.g., $\forall x_{in}$. Since requirements can be arbitrary complicated many formal theories have been developed to accommodate the expressiveness of English descriptions. This dissertation only considers state properties expressed as assertions which predicate ϕ must hold on all feasible program executions.

To summarize, in the scope of the dissertation a program P works as expected when P conforms to ϕ on any input to P – denoted $P \models \phi$. A straightforward way of evaluating whether $P \models \phi$ is by analyzing P (statically or dynamically) on every input and determining whether ϕ is violated on any of them. For a technique like testing that evaluates each input individually this verification approach is impractical since the size of the input domain can be arbitrary large. Fortunately, an alternative criterion exists which is based on a structural representation of P known as a control flow graph (CFG). The CFG of P is a directed graph where a node represent a basic block, i.e., a sequence of totally ordered statements in P, and edges are the possible flow of control between blocks. A path in a CFG represents a possible program execution. The CFG over-approximates the number of paths in a program because it does not account for potential dependencies between control flow branches. According to the structural adequacy criteria presented in [45] executing all feasible paths in the CFG is equivalent to executing P on all inputs. Hence an alternative evaluation of whether $P \models \phi$ is to determine whether ϕ holds on all feasible paths in the CFG.

The measurement describing how many paths have been validated, i.e., covered, is referred to as path coverage. In this dissertation, path coverage aimed at verifying ϕ is denoted as *path property coverage* (PPC). PPC characterizes the *degree* to which $P \models \phi$. Since many distinct inputs to P execute the same path, the number of inputs required to cover all feasible paths is usually far less than the size of the input domain. This fact makes the verification a more realistic task comparing to the exhaustive evaluation of P's input domain. Essentially, path coverage based verification partitions the input domain of P into equivalence classes with two domain elements being in the same class if they execute the same path. An equivalence class represents a partition of the input domain.

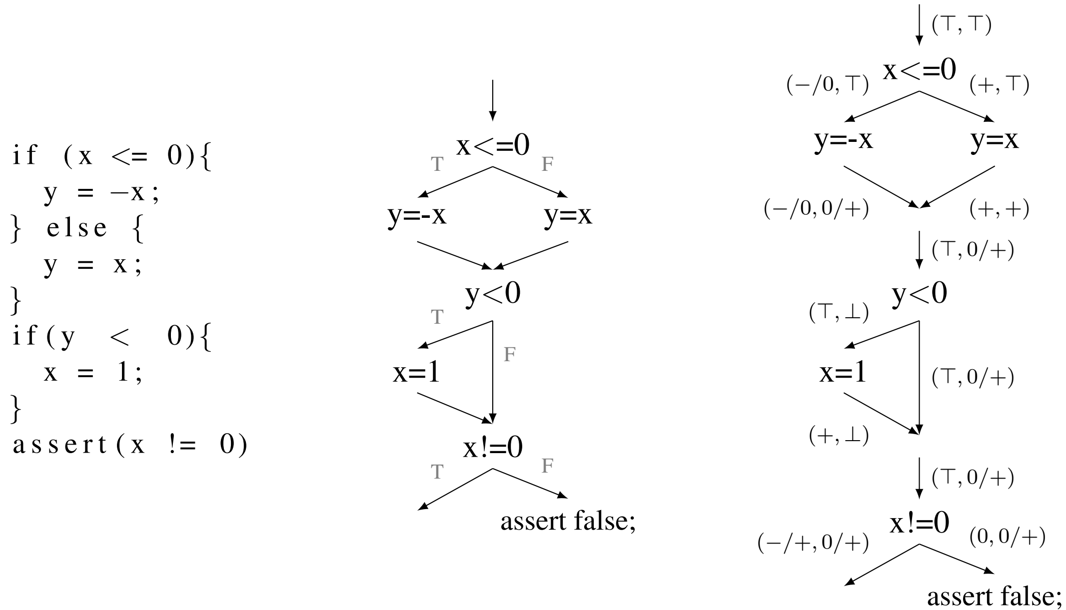


Figure 1.1: Partial result example: source code (left), CFG (middle), and the sign analysis result (right)

1.1.1 Dynamic Analysis

One widely used automated process for determining the degree to which $P \models \phi$ is testing. A single test, or a test case, consists of executing P on one of its inputs and validating that ϕ holds, e.g., for the Square Root example it would be running it with $x_{in} = 4$ and checking that $\phi == true$. Testing P on any element from a domain partition ensures the same result for any element of the partition. Therefore, knowing if two input values are in the same domain partition or not helps to minimize testing efforts.

A näive way of determining if two input values are in the same partition is to execute P on each of them and compare the traversed paths. An alternative approach is to characterize the partition as a logical formula over the P's arguments. Two input values belong to that partition if each of them satisfies the formula. Dynamically such a logical characterization of a path can be generated by *dynamic symbolic execution* (DSE) [40, 16]. In DSE a program is executed on a concrete value but alongside of the execution DSE tracks the

propagation of input values in a symbolic form and records the imposed constraints on these symbolic values. Execution of each conditional statement gives rise to a single constraint in the form of a predicate over symbolic values. The resulting conjunction of the generated constraints, called the *path condition*, characterizes the domain partition of the executed path. This implies that a domain partition can have a dual representation, one as a path in the CFG of P and another as a logical formula over P's input parameters.

To demonstrate the DSE concept consider the code on the left of Figure 1.1. In this and the following examples, the property, ϕ is embedded as an assertion into the code. The control flow graph (CFG) of this program (the middle of Figure 1.1) desugars the `assert(x!=0)` into an explicit branch that controls the execution of a false assertion when the condition (`x!=0`) fails to hold. The positive (true) and negative (false) outcomes of a branch are labeled with T and F, respectively. The concrete input of $(-1, 0)$ executes the path as presented on the left of Figure 1.2. The elided portions of the CFG are not part of the executed path and, thus, are not considered by the DSE runs. The remaining edges are annotated on both sides – the left side with the concrete values of `x` and `y` variables and the right side with the corresponding symbolic values and the constraints imposed on these values delimited by a comma.

Initially the values of `x` and `y` are presented by the symbolic values $\mathcal{X}$ and $\mathcal{Y}$, respectively, with no constraints associated with either of the symbolic values. On the input $(-1, 0)$ DSE executes the first branch and adds $\mathcal{X} \leq 0$ constraint to the path condition. After executing `y=-x` statement the symbolic value of `x` is negated and assigned to `y`. The building of the path condition proceeds in similar manner for the rest of the execution.

A dynamic analysis such as testing or DSE is inherently conditional, in the sense that its result holds when the input values satisfy the path condition, i.e., holds on a single path. In order to prove that $P \models \phi$ an analysis must determine that $P \models \phi$ for all domain partitions. A dynamic analysis may do this by accumulating the information from a set of

executed tests. Because of the duality in the domain partition representation, $P \models \phi$ holds when either all feasible paths in the CFG have been executed or the disjunction of path conditions becomes a tautology, i.e., the analysis becomes unconditional.

Nevertheless, using a dynamic analysis for program verification is a daunting task for two main reasons. One is that identifying infeasible paths in a CFG requires extra non-trivial effort like performing additional checks on path conditions of DSE. The second reason is the possibility of infinitely many paths in P on loops bounded by the input parameters. The presence of loops reduces the problem of covering all paths to the problem of executing P on all elements of its input domain. Therefore, for an arbitrary P verifying that $P \models \phi$ by means of testing is generally considered intractable.

1.1.2 Static Analysis

An alternative approach for deciding whether $P \models \phi$ is program verification. A verification technique analyzes program behaviors relative to ϕ statically, i.e., without executing the program. This dissertation explores static analyses, data-flow analyses in particular, as an instance of the program verification approach. The choice of data-flow analyses is explained by the restriction on the type of ϕ. Since ϕ is expressed as an assertion over program variables, the information propagated in a data-flow analysis may be used to reason about ϕ.

A static analysis interprets P's code by over-approximating the executable paths of P, i.e., besides feasible paths the analysis may examine infeasible paths as well. The primary advantage of the over-approximation of paths is that it enables a static analysis to reason about infinitely many paths. This power comes from the ability of static analyses to perform computations on sets of concrete values described by predicates, e.g., the $notZero \triangleq e \neq 0$ predicate describes all concrete values of the expression e that are not 0. The set of

predicates, commonly referred to as an *abstract domain*, is one of the defining components of a static analysis. Static analysis computes abstract values for each program statement and propagates these abstract values forward or backward through CFG edges. Static analysis continues with the computation until there is no change in the newly calculated abstract values, i.e., the computation reaches fix point.

To illustrate static analysis technique, consider the *sign* static analysis, SA_{sign}, that attempts to prove that ϕ holds on all paths of the CFG. This flow-sensitive analysis abstracts integer values according to their sign, i.e., whether they are less than (–), equal to (0), greater than zero (+), or their combination 0/+, –/0 or –/+. In general, the analysis approximates the sign of a value by recording a set of signs (a subset of $\{$–, 0, +, 0/+, –/0, –/+$\}$) for each variable; for notational convenience the set of all abstract values, $\{$–, 0, +, 0/+, –/0 ,–/+$\}$, is denoted $\top$, the top element, and the empty set of abstract values is denoted $\perp$, the bottom element. Essentially $\top$ represents any value of the concrete domain and $\perp$ implies that no such value exists in the concrete domain.

The right side of Figure 1.1 shows the results of the sign analysis. Compared to the original CFG the graph is annotated with the result of the analysis and contains two additional unlabeled nodes placed at the merge points of control flows. The edge from a merge node to the subsequent node is labeled with the joined abstract values of the merged control flows. In this example where ϕ is expressed as `assert(x=!0`.

There are two variables in this program, so the analysis records a pair of sets, (x, y), which is initially $(\top, \top)$, i.e., any value in the concrete domain. SA_{sign} propagates this initial values to the first conditional statement `x<=0` where it determines that on the false branch the values of `x` only can be greater than zero, i.e, +, while on the true branch its value only can belong to –/0 set. Since no actions have been performed on `y` its abstract value remains $\top$. At the assignment statement `y=x` the analysis assigns the abstract value of `x` to `y`. Similarly, SA_{sing} treats `y=-x` on the opposite branch. At the merge point of the

branches SA_{signs} combines abstract values of each variable by performing union operation on the corresponding sets, i.e., for `x` –/0 $\cup$ + $=$ $\top$ and for `y` 0/+ $\cup$ + $=$ 0/+. Thus the abstract values of the first merge point, i.e, before node `y<0`, over-approximates abstract values of the merged control flows, i.e., the value of `x` was –/0 on one incoming edge and + on another while the joined value of `x` is $\top$. When the concrete set representing a variable value increases, e.g., from + to 0/+, it is said that the analysis loses its *precision*. The analysis processes the rest of the program in the same manner.

In this example the analysis fails to verify $P \models \phi$ since both branches of the assertion statement are feasible, i.e., neither of the variables are assigned to $\bot$. When the abstract value of a variable is assigned to $\bot$ element it means that no concrete values exists for that variable, which means the execution leading to such abstract value assignment is infeasible.

When a static analysis determines that $P \models \phi$ then it means that ϕ holds on both feasible and infeasible sets of paths. However, the inability of a static analysis to distinguish between feasible and infeasible paths negatively affects its applicability when the analysis fails to prove that $P \models \phi$ as it happens in the example in Figure 1.1. This means that of the paths the analysis considered there exists at least one on which ϕ does not hold, but it is unknown if that faulty path is feasible or not. If the path is feasible then this is a true violation of ϕ. If the violation happens on an infeasible path then it is a false violation, i.e., false positive.

1.2 Partial Results of Static Analysis

To eliminate false positives verification can proceed further in one of two ways. The first approach is to identify a feasible path leading to the assertion violation and another is to enrich, i.e., refine, the set of predicates. For example, the abstract domain of $zero$ and $notZero$ may be partitioned further, i.e., refined, into $-$, 0 and $+$ sets. Unfortunately,

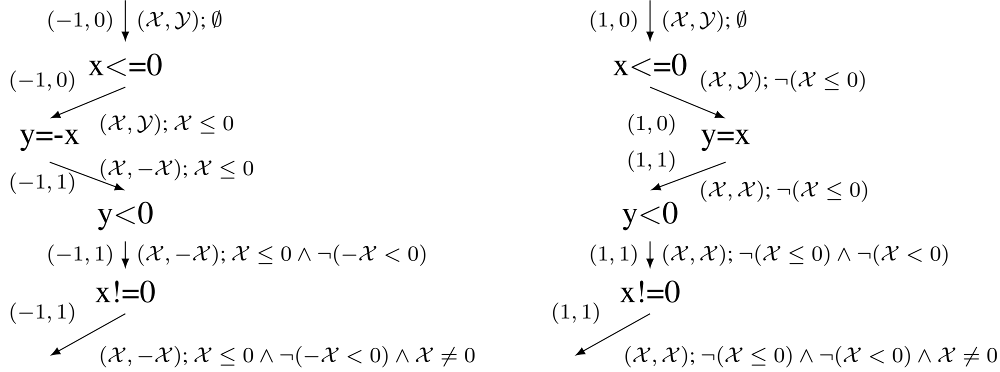


Figure 1.2: Partial result example: DSE on inputs $(-1, 0)$ (left), and $(1, 0)$ (right)

the information calculated by the original, i.e., failed, analysis is typically discarded by existing analysis techniques. Yet, this information might contain some partial results, like an infeasible branch of a conditional statement that may be useful. Such partial results imply that none of the paths in the CFG traversing that branch outcome violates ϕ. Hence, if present, this partial information contributes to the PPC of P.

To illustrate the concept of partial result of static analysis, consider the program example in Figure 1.1 where on the left is the code, CFG is in the middle and the result of SA_{sign} is on the right.

Even though SA_{sign} is not able to prove $P \models \phi$ since both outcomes of `x!=0` are possible, it can determine that the true branch of `y<0` branch is infeasible because one of the abstract values is $\bot$. Thus providing a partial output indicating that all paths in the CFG that traverse $(\top, \bot)$ edge are infeasible and thus vacuously do not violate ϕ. Such partial results imply that none of the paths in the CFG traversing that branch outcome violates ϕ. Hence, if present, this partial information contributes to the PPC of P. In order to evaluate the benefit of the partial information produced by a static analysis, its PPC contribution can be compared to the amount of work needed by another analysis, such as DSE to achieve the same PPC.

In order for DSE to achieve the same PPC it should execute paths that traverse all feasible prefixes leading to `y<0`. In the current example there are two prefixes – one with the positive and another the negative outcomes of `x<=0` branch. The concrete input of $(-1, 0)$ realizes the execution of the former and $(1, 0)$ input executes the latter. Figure 1.2 presents the results of DSE executions for each input.

In order to determine if the positive outcome of `y<0` is feasible, the last constraint of the path condition after `y<0` branch is negated for each DSE run, i.e., $\mathcal{X} \leq 0 \wedge (-\mathcal{X} < 0)$ for $(-1, 0)$ and $\neg(\mathcal{X} \leq 0) \wedge \mathcal{X} < 0$ for $(1, 0)$, and the modified path condition is checked for satisfiability. Clearly neither of the modified path conditions are satisfiable. Since on neither branches of `x<=0` the positive outcome of `y<0` is possible, DSE conveys that the true branch of `y<0` is never executed, which is exactly the same as the partial result of SA_{sign}.

Generally in order for DSE to prove that an outcome of a condition statement is infeasible it must explore all possible prefixes leading to that statement, which can be a large number especially if prefixes include loops. Moreover, if the information about an infeasible outcome has been calculated by a static analysis then there is no need to invoke a decision procedure for that condition statement in a DSE run.

The discussion above leads to the following research question:

> Q1: *How does the effort of generating PPC from partial information from a data-flow analysis compare to the effort required by ideal DSE runs to obtain the same PPC?*

1.3 Conditional Static Analysis

Over-approximation of program paths may cause static analyses to fail in verifying $P \models \phi$. The reason for this over-approximation is the requirement imposed on a static analysis to

express the value of a variable at each merge point of control flows as a single element of its abstract domain. At a merge point of control flows the static analysis combines the variables' abstract values propagated along the merging control flows. The combined abstract values are then expressed as a single element from the abstract domain. This merge operation may introduce imprecision into the static analysis result by over-approximating the result of the merged values.

One way to improve the precision of a static analysis is to allow it to operate on a subset of paths because the fewer paths the analysis considers the fewer merge operations it performs. The traditional way of restricting the set of paths to be analyzed is by assigning initial abstract values to the program arguments. For example, Cousout&Cousout in [14] incorporated an entry assertion on P's initial states into the definition of a program analysis. This restriction on entry values is also called a precondition. Similarly to dynamic analyses, the map between a precondition and the set of paths the precondition caused to analyze is not known a priori. Moreover, the map may be neither injective, i.e., each precondition maps to a unique set of path, nor surjective, i.e., there are paths that cannot be described by any precondition. This happens when the abstract domain elements cannot precisely describe the set of predicates that govern branching in P. If expressing an arbitrary set of paths through a precondition is impossible then imposing a restriction on P's arguments cannot guarantee an improvement in the analysis precision.

This dissertation proposes the use of explicitly defined sets of paths as an alternative precondition definition. In this approach a static analysis propagates its data-flow facts only through edges included in the path-based precondition. In the literature[8, 13, 32] such an approach is called conditional analysis since an analysis can determine that ϕ holds conditionally, i.e., only on some set of paths.

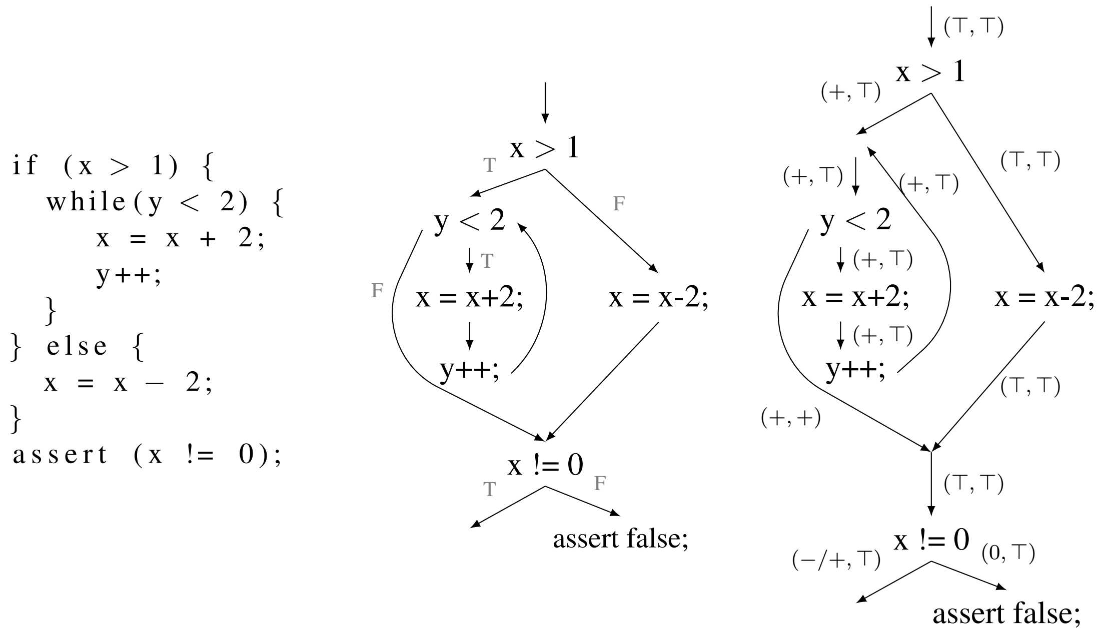


Figure 1.3: Conditional analysis example: source code (left), CFG (middle), and signs analysis (right)

To illustrate the idea of conditional static analyses consider a small fragment of code on the left of Figure 1.3. As in the previous example the middle of Figure 1.3 presents the CFG and on the right shows the results of SA_{sign}.

The results of SA_{sign} show that the abstract values $(\top, \top)$ reach the condition tested by the assert. This means that the assert condition may or may not be violated – the assert is not violated in this example on any program execution. Unlike the previous example where SA_{sign} was able to produce partial information, here the analysis fails to produce any PPC. Even in this simple example, it may take a minute to spot the fact that the source of the false violation report in the analysis is the inability to precisely analyze the false branch out of `x>1`. The source of this imprecision is the *misalignment* between the SA_{sign} input domain partition and the program predicates, i.e., using its abstract domain elements SA_{sign} has no means to accurately describe `x>1` or `y<2` predicates, thus it safely over-

approximates them by $\top$ element. This type of imprecision leads to false warnings and in larger programs, spotting false reports can be very time consuming [38].

Despite the fact that the original application of the sign analysis was ineffective the analysis can output PPC on a subset of paths. For example, restricting the set of paths analyzed by SA_{sign} to the paths for which `x>1` holds enables SA_{sign} to determine that the assertion is never violated on those paths. The left side of Figure 1.4 shows the result of the conditional SA_{sign} . Thus conditional static analysis may allow for sufficient precision to decide ϕ for that condition.

Naturally the PPC resulting from a conditional static analysis must be composed with the condition itself, i.e., if a conditional analysis determines that ϕ holds then it is only known to hold on the paths that the analysis has examined. Similarly to dynamic analyses in order for a conditional static analysis to determine whether $P \models \phi$ the results of the analysis with different conditions must be combined. However, unlike a dynamic analysis which always returns PPC, a conditional static analysis might not produce any new PPC comparing to its full version. For example, if the condition of SA_{sign} is the set of paths with the false outcome of `x>1` then no additional PPC is produced. The effectiveness of a partial static analysis is the subject of the second research question.

> Q2: *How much additional PPC does a conditional static analysis produce on a set of paths* Π' *compared to the PPC obtained by the same analysis on a larger set of paths* $\Pi \supset \Pi'$*?*

1.4 Combining Path Property Coverage

In the context of this dissertation a conditional static analysis can be directed towards a set of paths for which PPC has not been determined. A dynamic analysis can also be directed to traverse an unexplored path by executing the dynamic analysis on the concrete

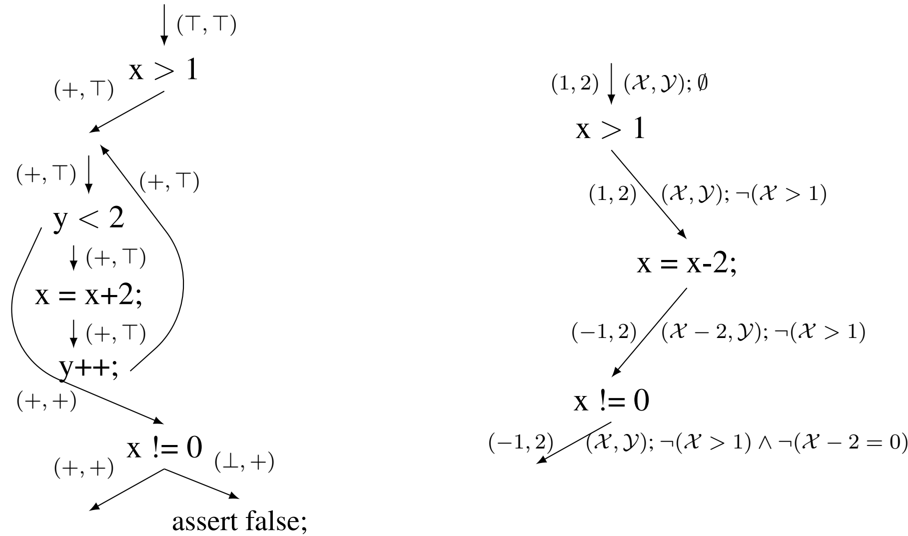


Figure 1.4: *Conditional* signs analysis (right) and test case with DSE (left) results

input values from a domain partition with unknown PPC. The strategy of finding a suitable condition depends on the analysis. For example, a set of paths analyzed by a conditional static analysis might be systematically decreased, e.g., split in half. While for DSE a new concrete value can be calculated by taking a previously explored path condition, negating one of its conjuncts and solving for the symbolic value the prefix up to the selected conjunct together and that negated conjunct. In the literature this dynamic technique of finding were inequivalent concrete values is known as *concolic execution* [40].

The execution of the code in Figure 1.3 by DSE on input $(1, 2)$ will result in the propagation of concrete values and symbolic constraints along the program execution path as illustrated on the right of Figure 1.4 where, as usual, the left side of an edge is annotated with the concrete values and its right side is annotated with symbolic variable values and path conditions. The information from this test case demonstrates that all program executions where $\neg(\texttt{x>1})$ holds are guaranteed not to violate the assertion (since the assert branch condition is unsatisfiable, $\neg(\mathcal{X} > 1) \wedge \mathcal{X} - 2 = 0 \equiv \bot$ – the infeasible outcome of the assert condition). However, due to the presence of the loop, any reasonable number of

test cases exploring the true outcome of `x>1`, is not able to provide a complete coverage of the program executions.

Regardless of the amount of time given to conditional SA_{sign} described in the previous section (displayed on the left of Figure 1.4) and DSE, their inherent weaknesses prevent either of the analyses to produce complete PPC for this example. However, their combined PPC verifies that $P \models \phi$ since DSE is able to prove it on one part of the CFG and SA_{sign} on its complement. Hence, the accumulation of path coverage from multiple analyses provides a more accurate assessment of the space of program behaviors that have been analyzed and found to satisfy the assertion.

To generalize the previous example, let each each of n distinct analyses accumulate within time t its own PPC_i^t $i \in \{1 \ldots n\}$. Obviously $\forall i$ $PPC_i^t \subseteq \cup_{j=1}^{n} PPC_j^t$. But more interesting is the question of whether the PPC_i^t from different analysis complement each other or are redundant. Since dynamic and static analyses are based on different techniques and static analyses have different abstract domains one might expect that there is little redundancy in PPC_i. In this case running each of the analysis t/n amount of time and combining their PPC might result in greater PPC than the PPC provided by any single analysis running for t amount of time, i.e., $\forall i$ $PPC_i^t \subset \cup_{j=1}^{n} PPC_j^{t/n}$. This conjecture is addressed by the third research question.

> Q3: *Given a fixed effort, how does the PPC of a single analysis compare to the combined PPC of a set of diverse analyses?*

1.5 Thesis Statement and Contributions

If the hypotheses posed in the research questions are found to hold, then they will support the overall goal of the dissertation:

> A unifying approach that incorporates a range of program analyses can efficiently produce greater behavioral coverage than analyses operating in isolation.

To answer the research questions and hence to support the thesis statement the dissertation work first needs to adequately pose and quantify these questions. Formalizing the notions of PPC is one of the conceptual units. Because some analyses can reason about infinitely many paths methods that employ counting and enumeration of PPC are not applicable. Thus in order to quantify and compare PPC of different analyses the dissertation develops an alternative structural description of path coverage. Moreover, since research on dynamic and static analyses emerged from separate research communities, each type of analyses has historically been viewed as a distinct framework. Thus, the essential building blocks in pursuing the thesis statement are adopting a unified view on different analyses and describing the analyses' behavioral coverage in a uniform way. In addition, to enable accumulation and sharing of behavioral coverage among analyses a unification framework is defined and implemented. The important properties of such a framework are to compactly yet descriptively represent behavioral coverage and to enable efficient querying of the stored behavioral coverage information.

The rest of the dissertation is organized as follows. The next chapter presents background information on program analyses and describes related work on partial analyses and on combining static analyses. The formalism presented in that chapter lays necessary foundations for the rest of the dissertation. Chapter 3 includes the overall vision on the behavioral coverage framework. Path property coverage is presented in Chapters 4. The posed research questions and the thesis statement are addressed in the penultimate Chapter 6. The last chapter presents the conclusion and elaborates on future work.

Chapter 2

Background and Related Work

In order to streamline the explanation of the related work and the novel analysis unification framework the dissertation first presents relevant background information for several dynamic and static program analyses. The main differences between these two analysis techniques is that dynamic analysis is performed during program execution while static analysis interprets the source code of P. Here dynamic program analysis is exemplified by testing and dynamic symbolic execution techniques which are used in the dissertation experiments. The static analysis section consists of an explanation of data-flow analysis and information on its variations: predicate abstraction and logical interpretation. These variations provide more precise analysis than traditional data-flow static analysis.

This chapter proceeds by summarizing previous work on combining analysis as another technique that increases the precision of the cumulative analyses. The section on combining analysis highlights the differences between combining intermediate results and final results of program analyses. This dissertation focuses on combining final results of different analyses which evaluate a program assuming different conditions. The section on conditional analysis describes related work that seeks improvement in precision when only part of a program's state space is explored and letting the same, or another analysis, complete the verification of the remaining state space. The remainder of the chapter briefly explains how the idea of conditional analysis has been extended to data-flow analysis.

Since every analyses technique discussed here utilizes a graph representation of P's structure the definition of Control Flow Graph (CFG) [1] is given first.

Definition 2.0.1 (CFG). *CFG is a directed graph where a node l is a statement, and an edge (l, l') represents the control flow between statements l and l', i.e., the order in which statements are executed.*

A conditional statement has two outgoing edges, i.e., branches,: one to the statement executed on its true outcome and another to the statement executed on its false outcome.

Since the outgoing edge of the last statement in the body of a loop points back to the loop's condition it creates a cycle in CFG. A CFG has one root node, i.e., the entry point for P, and one or more leaf nodes, i.e., the termination points for P. A CFG represents all paths that P may execute.

2.1 Dynamic Program Analysis

2.1.1 Testing and Adequacy Criteria

A single test case consists of two parts: an execution of P on concrete input values and the evaluation of the execution for conformance with ϕ. Running a finite set of test cases, called a test suite, comprises testing. Since exhaustive testing, i.e., testing the whole input domain, is prohibitively expensive a decision must be made on what test cases to include into a test suite. There are two distinct approaches for test case selection. One of them takes into account the internal structure of P and the other does not. The former is commonly referred to as a white-box or design-based testing while the latter is called a black-box or requirements-based testing [28]. Since the proposed framework exploits the coverage information of paths in P we discuss white-box testing in more detail.

In white-box, or *structural*, testing the quality of a test suite is quantified by the number of CFG elements such nodes, edges or paths, it was able to execute, i.e., cover. The intuition is that coverage of an additional element of a CFG requires a test case that traverses a different path through the CFG, thus potentially exposing additional errors [7]. Moreover, structural testing allows for exhaustive coverage of at least some structural elements in P's CFG [35]. The requirement for complete coverage of feasible CFG elements, such as nodes or edges, is called an *adequacy criterion* and is used to guide the construction of a test suite [45]. When a test suite meets the objectives for an adequacy criterion it is said to

be adequate, i.e., tested appropriately, for this criterion. The commonly known structural adequacy criteria are method, statement, branch and path. These criteria generally require different numbers of CFG paths to cover their elements. The larger the number of paths are covered by the criterion adequate test suite the higher confidence that $P \models \phi$. The strength of a criterion corresponds to the size of its adequate test suite. Therefore criteria have different strength of determining that ϕ holds in P. Various structural criteria can be organized into a partial order according to their strength with the *path-adequacy* criterion being the maximum element [21].

The monitoring of coverage information is generally done through program instrumentation but it also can be implemented in the execution environment of P, e.g., Java Virtual Machine (JVM). An instrumentation-based coverage tool analyzes P's code to construct a CFG and to instrument P with additional code. During P's run the inserted code communicates to the coverage tool what statements have been executed. Using this information and the previously constructed CFG the tool then produces cumulative coverage information. In practice, coverage tools usually track weaker coverage elements like class, method and statement, e.g., EMMA [37], the state of practice tools track branch coverage, e.g., gcov [22] and Cobertura [18], and almost none handle path coverage.

2.1.2 Dynamic Symbolic Executions

Symbolic execution (SE) [31] interprets P on symbolic input values instead of concrete ones. During the interpretation of P, symbolic execution denotes the values of program variables as expression over symbolic inputs. Using the discussion in [31] the notion of SE can be defined as

Definition 2.1.1 (Symbolic execution)**.** *An alternative semantic for a programming language where the real data objects can be represented by arbitrary*[1] *symbols. Such computation accepts symbolic inputs and produces symbolic formulas as outputs.*

When symbolic execution encounters a conditional statement it generates boolean expressions, i.e., conditions, for each branch of the statement. Each path in the CFG is associated with a distinct list of conditions. The conjunction of these conditions is called a *path condition* – pc. Thus pc is a Boolean expression over symbolic inputs that uniquely describes a path in the CFG. On entry to P the value of pc is set to $true$ and at a conditional statement with the symbolic expression q, pc is extended for the true branch to $pc \wedge q$ and for the false branch to $pc \wedge \neg q$. The symbolic execution for each branch then proceeds independently. A state of a symbolically executed program contains the mapping of program variables to their symbolic expression and also the current pc. Thus for the program on Figure 1.3 the symbolic state after `x=x-2` is $\sigma = (x \mapsto \mathcal{X} - 2, y \mapsto \mathcal{Y}; \neg(\mathcal{X} > 1))$, where pc is $\neg(\mathcal{X} > 1)$ and symbolic expression of `x` and `y` are $\mathcal{X} - 2$ and $\mathcal{Y}$ respectively, with $\mathcal{X}$ and $\mathcal{Y}$ being symbolic input values.

Since SE interprets P statically it may encounter cases when the pc is unsatisfiable because of the presence of data dependencies that make some branches infeasible. In order to determine if the constructed pc that includes the condition for a new branch is satisfiable or not, a symbolic execution formulates the pc as a query to a constraint solver. If the pc is unsatisfiable then symbolic execution abandons that branch. For the same example on Figure 1.3 consider a path that traverses the false branch of `x>1`. Before evaluating the `x!=0` condition the path has $pc = \neg(\mathcal{X} > 1)$ and the variable `x` is mapped to $\mathcal{X} - 2$. When interpreting `x!=0` SE generates new pc by extending the current pc in two ways. One adds the false branch $pc_f = (\neg(\mathcal{X} > 1) \wedge \mathcal{X} - 2 \neq 0)$ and another adds the true branch

[1] In this context arbitrary means unconstraint value of the object type

$pc_t = (\neg(\mathcal{X} > 1) \wedge \neg(\mathcal{X} - 2 \neq 0))$. A solver determines that pc_t is satisfiable while pc_f is not.

As with any path-sensitive analysis symbolic execution can run indefinitely in the presence of loops. In order to guarantee the termination of symbolic execution a bound, k, is set on the maximum number of times it can unroll a loop. For the same program example if $k = 0$ then symbolic execution generates two additional paths $(\mathcal{X} > 1 \wedge \neg(\mathcal{Y} < 2) \wedge \neg(\mathcal{X} \neq 0))$ and $(\mathcal{X} > 1 \wedge \neg(\mathcal{Y} < 2) \wedge \mathcal{X} \neq 0)$.

As discussed in Section 1.1.1 dynamic symbolic execution (DSE) [40, 16] is a combination of testing, i.e., concrete execution, and symbolic execution. Similar to testing a program is executed on concrete values but the alongside of the concrete execution symbolic execution is performed as well. DSE collects pc along the executed path thus logically expressing a set of input values that execute the same path. The main difference between symbolic and dynamic symbolic executions beside the number of paths they analyze is that the former requires calls to a solver while the latter does not because it only follows feasible paths. The following definition of DSE is compiled from [40, 16]

Definition 2.1.2 (Dynamic symbolic execution)**.** *Simultaneous realization of concrete and symbolic executions of a program on a concrete input.*

One extension of DSE, called *concolic testing* [40], is used for test case generation. After executing an initial test case and obtaining its $pc = q_1 \wedge q_2 \wedge \cdots \wedge q_n$ each condition q_i is systematically negated to produce a new Boolean expression $pc_i = q_1 \wedge q_2 \wedge \cdots \wedge \neg q_i$, $i \in \{1 \ldots\}$. Next the constraint pc_i is passed to a solver to find an assignment to symbolic values that satisfy that constraint. If a solution exists then a new test case is created otherwise pc_i is said to be infeasible. The dissertation uses DSE to collect a characterization of the program behavior that is covered by a test suite.

2.2 Static Program Analysis

2.2.1 Data-Flow Analysis

A common class of static analyses are the flow-sensitive data-flow analyses which calculate facts about the flow of data values along program paths [33]. There are significant variations among data-flow analyses. They typically differ in the facts that they compute, e.g., they may approximate values of selected program variables or relations among variables, such as, data dependence analyses. Another variation in the analyses deals with whether the computed facts at particular program locations hold on all CFG paths leading to that point or on at least one of such path. Since ϕ is assumed to be a universal property the dissertation considers only analyses that determine facts that hold on all program paths.

In any case, a data-flow analysis computes the set of facts by traversing a CFG. Section 1.1.2 demonstrated an instance of data-flow analysis. For each node in the CFG, the fact on the exit of the node, i.e., after the interpretation of statements corresponding to the node, is computed as a function of the facts coming from all entries to that node, i.e., after the interpretation of all predecessor nodes. This computation is captured by two types of equations.

The first equation takes the incoming facts of the node and merges them together into a single entry fact. Depending on the analysis, its merge function can combine incoming facts such that they hold for all paths leading to the node or for at least one path. This equation for a node s is expressed as

$$in_s = merge_{p \in pred(p)}(out_p)$$

where out_p are the exit facts of the predecessors p of s in the CFG and in_s is the result of the merged out_p facts.

The second equation produces the exit facts of the node by propagating the in_s through statements associated with s and modifying facts according to the rules for s, e.g, removing elements from the entry set and/or adding elements to the entry set. This equation is expressed as

$$out_s = prop_s(in_s)$$

The function $prop_s$, a *transfer function*, is constructed for each combination of s and in_s, i.e., it is parametrized by the node type and the fact type.

Since the facts on the exit of one node becomes the entry facts of another node, the changes in the former would require the recalculation of the latter. In the presence of loops in P, this may require iterative calculation that continues until a fixed point is reached.

Reaching a fixed point is an important requirement for a data-flow analysis since it guarantees analysis termination. This requirement restricts the choice for the types of facts that analyses can propagate. In particular, the set of facts must be organized into a partially ordered set satisfying the ascending chain condition. The set of facts comprises an *abstract domain* D and the partial relation between elements of D is presented as a partially ordered set, i.e., a lattice, L_D [33]. The ascending chain condition guarantees that during iterative merges the size of facts will not grow infinitely and eventually will stabilize. Any complete lattice satisfies that condition.

The following formal definition of a static analysis $\mathcal{A}$ is the same as presented by Flemming Nielson et al., [33]:

Definition 2.2.1 (Data-flow analysis)**.** *Given:*

- *The complete lattice $D_{\mathcal{A}}$ that describes the abstract domain of $\mathcal{A}$.*
- *CFG_P for a program P.*

- *A set of monotone transfer functions $\mathcal{F}_{\mathcal{A}}$ for each statement $l \in CFG_P$ that maps an element of $D_{\mathcal{A}}$ to itself, i.e., $f_l \in \mathcal{F}_{\mathcal{A}} : D_{\mathcal{A}} \mapsto D_{\mathcal{A}}$.*

- *Entry statements E in CFG_P.*

- *An initial value $\iota \in D_{\mathcal{A}}$ for statements in E.*

Then the set of equations for forward $\mathcal{A}$ is defined as follows on entry and exit of each statement $l \in CFG_P$:

$$
\begin{aligned}
\mathcal{A}_{in}(l) &= \bigsqcup \{\mathcal{A}_{out}(l') \mid (l', l) \in CFG_P\} \sqcup \iota_E^l \\
&\quad \textit{where } \iota_E^l = \begin{cases} \iota & \textit{if } l \in E \\ \bot & \textit{if } l \notin E \end{cases} \\
\mathcal{A}_{out}(l) &= f_l(\mathcal{A}_{in}(l))
\end{aligned}
$$

where $\sqcup$ is the least upper bound operator, $\bot$ is the bottom element of $D_{\mathcal{A}}$ for which $\forall d \in D_{\mathcal{A}} : \bot \sqcup d = d$ and $\forall l \in CFG_P : f_l(\bot) = \bot$. For the safety properties $\bot$ corresponds to the empty set of concrete values and $\top$ to the set containing all concrete values. The value of ι is assigned to $\top$, i.e., the analysis considers all possible input values for a program. The solution of the above set of equations provides is the result of the analysis for P.

Since in the context of the dissertation ϕ is expressed as a predicate over program variables the facts for analyses considered here must describe program states, i.e., mappings between the program variables and their values. Those facts are referred to as *property facts* since deciding $P \models \phi$ depends on them. Property facts represent an *intermediate result* in the sense that they cannot reason about ϕ directly. In order to determine $P \models \phi$, the static analysis is augmented with a results decision, which is specifically designed to

reason about ϕ by examining the computed property facts. This results decision is what produces the *final result* of static analysis. For example, the intermediate result of SA_{sign} on Figure 1.1 are the labels of CFG edges and the final result of SA_{sign} is the output that the true branch of `y<0` is infeasible.

Different static analyses can be instantiated from the same analysis framework by varying the abstract domain D of the framework. Since both transfer and merge functions take the elements of abstract domain as arguments, changes in D requires the modification of those functions. The finer the partitions of the concrete domain described by the abstract domain D, the more precise the corresponding static analysis. Thus for the lattice L_{zero} depicted on the left of Figure 2.1 the static analysis framework instantiates SA_{zero}, while for SA_{sign} from the previous chapter is instantiated by L_{sign} that is on the middle of the same figure. In this case SA_{sign} is said to be more precise than SA_{zero} since the former analysis can represent finer sets, e.g., non-positive integer values, which cannot be described by the elements of L_{zero}. Obviously, a static analysis can achieve the utmost precision when its abstract domain is the power set of concrete integer values, i.e., $L_{\mathcal{P}(\mathbb{Z})}$, since such domain has an element that exactly describes any set of values that a variable can take on during program executions. However, as depicted on the right of Figure 2.1 $L_{\mathcal{P}(\mathbb{Z})}$ does not satisfy the ascending chain condition, i.e., the sets can increase indefinitely. Thus $L_{\mathcal{P}(\mathbb{Z})}$ cannot be used to instantiate a static analysis. In general, a static analysis may not use, for D, the concrete values computed by the program. Instead it uses an element that safely over-approximates values that leads to the imprecisions in property facts calculations.

2.2.2 Predicate Abstraction

Usually D of a static analysis is fixed permanently and the analysis applies the same reasoning to all programs. However, the expressiveness of elements of D may not be sufficient to

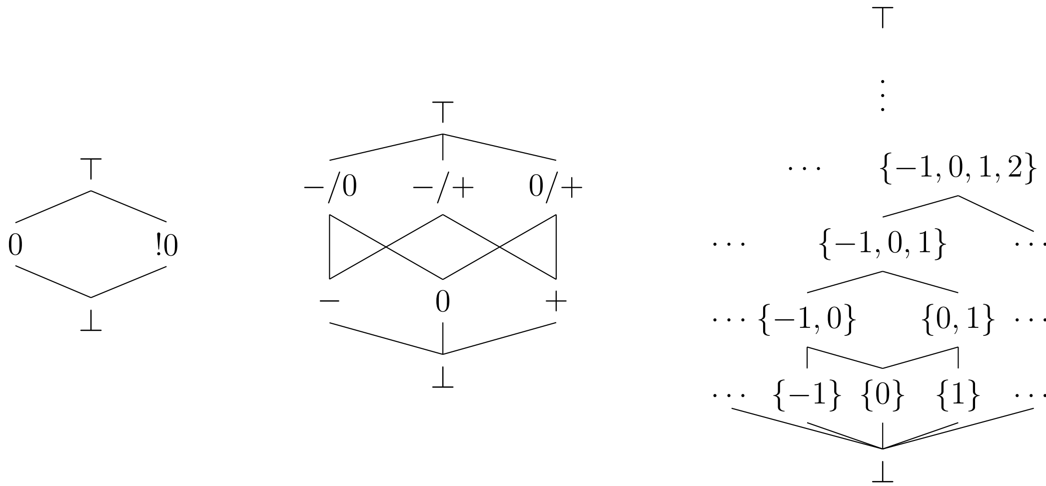


Figure 2.1: Lattices for $zero$ domain (left), $signs$ domain (middle) and $\mathcal{P}(\mathbb{Z})$ domain (right).

describe the predicates in P. For instance, it is reasonable to use the L_{zero} abstract domain if the predicate of conditional statements in P are of the form `x==0` or `x!=0`. However, if P contains only predicates of form `x<1` then using SA_{zero} will approximate the value of `x` with $\top$ on both branches which leads to an empty final result. But if D is designed to contain elements representing P's predicates then the precision of such analysis can be greatly improved over static analysis with a fixed abstract domain.

This is exactly what the *predicate abstraction* [24] technique does. Predicate abstraction partitions the concrete set of values into abstract sets that is suited for a particular P. In this approach P is analyzed first to determine suitable elements of D. Besides regular partitions described by predicates with a single variable, e.g., `x==0`, D of predicate abstraction can contain predicates that characterize the relation between variables, e.g., `x<y`. This feature makes predicate abstraction particularly useful in finding the loop invariants since they commonly consist of several program variables. The key challenge for predicate abstraction technique is to identify such predicates [20].

2.2.3 Combining Program Analysis

Refining the abstract domain of static analysis may not increase the precision of the analysis. It may happen that the values of one set of variables can be precisely expressed through one abstract domain while the values of another set of variables do not have exact representations in the same domain. For example, no elements of D_{sign} can express sets of even or odd values. Likewise, no elements of $D_{parity} = \{\bot, even, odd, \top\}$ can express positive or negative values. In order to increase the precision, SA_{sign} and SA_{parity} need to be made into a single analysis where the new abstract domain is the combination of D_{sign} and D_{parity}.

The classic work on combining program analysis is presented in the work of Patrick Cousot et al, [14] where the authors described three possible unifications of abstract domains. The first is the *direct product* of two domains which produces the same result as when the two analyses are run independently of each other. This type of combination does not increase the precision of the analysis and is considered to be trivial.

The next type is the *reduced direct product* where the elements of the new domain with the same semantic are merged into one. For instance the element $(0, odd)$ of the direct product $D_{sign} \times D_{parity}$ has the same semantic as $(\bot, \bot)$ element because 0 is an even number. In a similar fashion transfer functions are redesigned to include the reasoning between the domains. For instance, $(+, odd) - 1 = (\{0, +\}, even)$.

The third type of analysis combination is described by the *reduced direct power* which is a set of all isotone maps from elements of the first domain to the elements of the second domain. Formally the reduced power $L_{D_2}^{L_{D_1}}$ is the set of all isotone maps from D_1 to D_2 with $f \sqsubseteq g$ if and only if $\forall x \in D_1\ f(x) \sqsubseteq_{L_{D_2}} g(x)$. Essentially, the reduced direct power of two abstract domains allows for the interaction between variables with different abstract domains. For example when the abstract value of one variable depends on the evaluation

of the abstract value of another variable, e.g., for `boolean b=x>0` when x $\mapsto +$ then b $\mapsto true$.

Examples demonstrating the implementation of static analysis combination using the reduced direct power can be found in work by Cliff Click et al., [12] and Jeffrey Fischer et al., [19]. They clearly demonstrate that combining different analysis requires non-trivial effort. The main drawback of the reduced direct product or reduced direct power is that they demand a complete rewriting of the previous analyses which is a tedious, error-prone and labor intensive work. This shortcoming has prompted several researches to look for a ways to automatically combine analyses in an efficient way using some general analysis assembler. This issue has been addressed by Patrick Cousot et al., [14] which stated that in general, it is not possible to design such an assembler and any attempt to automate the combination of analyses will produce sub-optimal results.

Yet, there may exist special classes of program analysis for which automated optimal combination is possible. The work on *logical interpretation* [26, 43] explored these possibilities. Logical interpretation aims to identify static analyses whose merge and transfer functions can be expressed as Satisfiability Modulo Theories (SMT) problems. Thus an analysis is designed using some theory and implemented with an SMT solver as the analysis engine. The abstract domain of logical interpretation consists of formulas over the theory's atoms and the program variables. The relation of the domain elements is done not by the inclusion relations but by implication relations between formulas. The elements of the logical lattice are computed online during analysis. The advantage of logical interpretation based analyses is the ability to automatically combine these analyses using techniques developed in SMT community. However, not all program analyses can be instantiated or combined using logical interpretation framework. For example, logical interpretation can only combine analyses which are described by disjoint theories, i.e., theories that only share the equality symbol.

So far the combination of program analyses, or combining analyses, has focused on combining the intermediate results of analyses to achieve better precision for deciding $P \models \phi$, i.e., the final result. Where intermediate results can be viewed as invariants calculated by a static analysis while the final result is the decision made by the analysis based on the invariants. However, in the literature combining analysis also can imply the combination of the analyses final results. For example in the paper by Shay Artzi et al., [4] the authors combined the final results of several analyses that determines whether the parameters of a method are immutable. In that work if one analysis determines that a parameter is mutable or immutable then the next analysis does not analyze that parameter. Similarly, the dissertation focuses on *combining final results* of different program analyses that analyze programs under different conditions. The next section describes the related work of conditional analysis technique.

2.3 Conditional Program Analysis

Previous research has introduced two "conditional" notions – *conditional soundness* and *conditional analysis*. In their essence the two concepts are related – the result of an analysis holds only under some conditions. They deviate in the approaches that they use to obtain the unconditional result. Conditional soundness relies on another analysis to prove that the program states that do not satisfy the condition are not reachable, i.e., the program executions containing those states are infeasible. While conditional analysis may use the same or a different analysis to show that those unsatisfying states do not violate properties by appearing in program executions that either conform to the properties or are infeasible.

The work on conditional soundness presented by Christopher Conway et al., [13] explored the dependency of the soundness of one analysis on the result of another analysis. Many analyses assume that a program is "well-behaved". For example SA_{sign} assumes

the absence of overflow conditions when the rule $+$ "plus" 1 produces not expected $+$, but $-/+$. The authors demonstrated this concept on a pointer analysis for C program. A points-to analysis can produce sound results if the program is memory safe, i.e., there is no out of bound memory accesses. The authors argued that it is possible to combine points-to and memory safety analyses into a single analysis but such an approach is not scalable. Thus they proposed to first perform points-to analysis assuming that the program is memory safe and after that run the memory safety analysis with the points-to analysis result to prove that the assumption holds.

Besides explaining the concept of conditional soundness the authors also formalized it through a set of definitions. They specified a conditional soundness with respect to a predicate θ on a concrete state. Then an analysis is defined to be θ sound when it over-approximates program behavior reachable through concrete states satisfying θ. Thus, an analysis applies its transfer function only to the states satisfying θ. However, the transfer function can produce non-θ states, but those states will not be explored on the next application of the transfer function. In order to prove unconditional soundness of the first analysis, the second analysis explores the states produced by the first analysis and proves that the resulting non-θ states are unreachable in the program.

The work by Mayur Naik et al., [32] on static race detection for multi-threaded programs is conceptually similar to the previous paper on conditional soundness. A static race detection algorithm analyzes memory accesses m_1 and m_2 guarded by locks l_1 and l_2 respectively. It assumes that m_1 and m_2 may alias and thus the algorithm aims to prove that l_1 and l_2 must alias in order for a program to be race free. The alternative approach proposed in this paper assumes that l_1 and l_2 must not alias and then it is left to show that m_1 and m_2 cannot refer to the same memory location. The absence of aliasing between m_1 and m_2 is easy to show when m_i is the field of the object referred by l_i and that the field m_i

is only reachable through that object. However, in order to make the result of the analysis unconditionally sound another analysis must be prove that l_1 and l_2 in fact must not alias.

Dirk Beyer et al., in their paper on conditional model checking [8] emphasized the fact that when a model checker fails either to verify a program or to produce a counterexample then the effort put into the exploration of the program's state space is wasted. These situations arise when either the model checker runs out of computational resources or its unable to decide on the predicate of a branch condition. The latter takes place when the model checker is not designed to handle the theory used in the predicate, e.g., the predicate analysis cannot handle non-linear arithmetic.

In the case of resource exhaustion, i.e., either running out of memory or timing out, instead of conventional termination the authors proposed an informative termination whereupon shutting down the model checker produces data that describes the parts of the state space it was able to verify. To characterize the parts of the analyzed state space the model checker enhances each abstract state with a predicate that uniquely describes that state. Hence, when the model checker depletes its computational resources, instead of terminating with a run-time exception, it returns a condition describing the set of states it has analyzed. The condition is expressed as a conjunction of the analyzed state predicates. A subsequent run of a model checker can use the negated condition to guide itself to the part of the state space that was not analyzed by the previous model checker. This characteristic of conditional model checker corresponds to the notion of conditional analysis.

In the case of undecidable predicates the authors proposed to proceed with such predicate interpretation that permits the model checker to proceed with the verification, e.g., assuming that an assertion it cannot reason about does hold. The encountered assumptions for taken branches are encoded as an automaton that is used to guide the further analysis to the assumed branches. These assumptions must be analyzed further by another model checker that is capable to reason about them. If the subsequent model checker determines

that these assumptions hold then the verification is complete. This side of conditional model checker relates to the idea of conditional soundness.

A conditional model checker integrates assumptions, i.e., conditional soundness for certain states, and predicate, i.e., conditional analysis, information into the characteristic of abstract states. At the end of the execution the conditional model checker traverses the reached abstract states and produces the condition and the branch outcome assumption information. The experiments presented in the paper have shown that applying conditional model checkers with feedback capabilities increased the number of programs that can be verified.

2.3.1 Conditional Data-flow Analysis

The dissertation extends the notion of conditional analysis to data-flow analysis. The requirement to analyze all paths can lead to a loss in precision due to the computations in $merge$. This can be avoided by forcing the analysis to consider fewer predecessor statements at merge points. This can be achieved by constraining the set of paths a static analysis examines. If the static analysis was able to decide $P \models \phi$ on that set of paths then the result is conditional and the condition is a predicate that describes that set of paths.

The conditional static analysis A^θ is described by its condition θ which can be used to identify the set of paths to be analyzed. In this work θ is represented by the set of branch edges in CFG_P, at most one for each conditional statement l, which the analysis must include while excluding their counterparts. If l has l' and l'' as its true and false targets, respectively, then θ can contain the edge (l, l'), or the edge (l, l''), or none of them. To capture the relation between the opposite branches of l let $(l, l') = \neg(l, l'')$ and the vice versa $(l, l'') = \neg(l, l')$. If $(l, l') \in \theta$ then the values of all variables x_i incoming to the target of its opposite edge l'', i.e., along edge $\neg(l, l')$, will be set to $\bot$. For brevity $\forall i : x_i = \bot$

is denoted as $\perp$ state. The same principle applies when $(l, l'') \in \theta$. When none of the edges are present in θ then the analysis treats them in its usual manner, i.e., propagates the information through both branches.

Thus a traditional data flow analysis is a conditional analysis with $\theta = \emptyset$, i.e, $\mathcal{A} = \mathcal{A}^{\emptyset}$. In the related work on the conditional model checking a condition θ also consists of "path" part. In the case when a model checker cannot reason about some set of paths it records them in a structural way which is similar to θ path-based condition.

With the proposed path-based of the constraint θ a conditional analysis $\mathcal{A}^{\theta}$ can be defined as;

Definition 2.3.1 (Conditional data-flow analysis)**.** *Given:*

- *The complete lattice $D_{\mathcal{A}^{\theta}}$ that describes the abstract domain of $\mathcal{A}^{\theta}$.*
- *CFG_P for a program P.*
- *A set of monotone transfer functions $\mathcal{F}_{\mathcal{A}}$ for each statement $l \in CFG_P$ that maps an element of $D_{\mathcal{A}^{\theta}}$ to itself, i.e., $f_l \in \mathcal{F}_{\mathcal{A}^{\theta}} : D_{\mathcal{A}^{\theta}} \mapsto D_{\mathcal{A}^{\theta}}$.*
- *Entry statements E in CFG_P.*
- *An initial value $\iota \in D_{\mathcal{A}}$ for statements in E.*
- *Set of edges θ to be excluded from CFG_P.*

Then the set of equations for forward $\mathcal{A}^\theta$ is defined as follows on entry and exit of each statement $l \in CFG_P$:

$$\begin{aligned}
\mathcal{A}^\theta_{in}(l) &= \bigsqcup\{\mathcal{A}^\theta_{out}(l') \mid (l', l) \in CFG_P, \neg(l', l) \notin \theta\} \sqcup \iota^l_E \\
&\quad \textit{where } \iota^l_E = \begin{cases} \iota & \textit{if } l \in E \\ \bot & \textit{if } l \notin E \end{cases} \\
\mathcal{A}^\theta_{out}(l) &= f_l(\mathcal{A}^\theta_{in}(l))
\end{aligned}$$

where for the first equation we assume $\forall d \in D_{\mathcal{A}^\theta} : \emptyset \sqcup d = d$. Using the condition θ and CFG_P one can determine the set of path Π that the conditional static analysis (CSA) examines.

Chapter 3

Unification Framework

One of the contributions of the dissertation is the unification framework that combines the *final results* of multiple program analyses that test or verify properties expressed as predicates over program variables at specific locations in the code. The development of such framework requires an abstracted view of program analyses that dissociates their internal implementations. The goal of the next two sections is to present a generalized analysis concept and discuss how the analyses described in the previous chapter can be instantiated from it. The presented abstracted view of analyses is relevant to the unification framework and might not, in general, be applicable for other purposes.

3.1 Generalized View of Analysis

One common factor among analyses is that they analyze program's behaviors for some part of its input domain which can be expressed as a set of predicates over input variables or as a set of CFG paths. In program testing, expressing the extent to which a program's executions have been analyzed is typically done indirectly, and weakly, using some syntactic coverage metric, e.g., the percentage of statements executed, and not in terms of the input domain that has been considered. If assertions are considered, whose predicates create branching within the programs flow of control, then path coverage produces the finest distinction between input domains that is needed to reason precisely about property satisfaction, i.e., verification, or falsification, fault detection. For this reason, in the unification framework the implicit or explicit path-based Π representation is chosen to describe the input domain.

Analyses vary in the sets of CFG paths, $\Pi \subseteq \Pi_{CFG}$ that they are able to analyze, where Π_{CFG} denotes all paths in the CFG. A single test or dynamic symbolic execution (DSE) is only capable of analyzing a single path $\pi \in \Pi_{CFG}$. Symbolic execution (SE) and conditional static analysis (CSA) are able to analyze a subset of paths, i.e., $\Pi_{SE} \subseteq \Pi_{CFG}$ and $\Pi_{CSA} \subseteq \Pi_{CFG}$ respectively. The traditional static analysis (SA), i.e., as in

Definition 2.2.1, considers all CFG paths, i.e. $\Pi_{SA} = \Pi_{CFG}$. Since SA is an instance of CSA with $\theta = \emptyset$, i.e., all CFG paths are analyzed, the dissertation refers CSA with such conditions as SA.

Obviously, differentiating program analyses by the set of paths they analyze cannot unambiguously identify an analysis. For example, a single path can be analyzed by CSA and DSE. In the previous chapter it has been discussed that the data-flow framework can be instantiated to produce different analysis by considering various abstract domains which determine the precision of the analysis. Similarly, the abstract domain can also be used to describe testing, DSE and SE. The abstract domain of testing is the concrete domain $D_c \equiv \mathcal{P}(\mathbb{Z})$, where as before $\mathcal{P}(\mathbb{Z})$ is the power set of integer values. DSE and SE use an abstract domain with symbolic precision D_s which consists of any symbolic expression that can be encountered during program executions. Depending on the program, the element of D_s can be different while its expressive power, i.e., the precision, stays the same.

To illustrate an instance of D_s for DSE consider the code below.

```
   void m (int x) {
1:   if (x < 4) {
2:      x++;   }
3:   x = 2*x;
   }
```

When DSE executes the code on input 0, it will traverse $\pi_{x=0} = 1 \rightarrow 2 \rightarrow 3$ path. Let x_n denotes the variable x after the statement n. Then, computed by DSE, the symbolic values

for x_n along every path comprise the domain of x:

$$D_s = \left\{ \begin{array}{l} \bot, \\ \mathcal{X} < 4 \wedge x_1 := \mathcal{X}, \\ \mathcal{X} \geq 4 \wedge x_1 := \mathcal{X}, \\ \mathcal{X} < 4 \wedge x_2 := \mathcal{X} + 1, \\ \mathcal{X} < 4 \wedge x_3 := 2(\mathcal{X} + 1), \\ \mathcal{X} \geq 4 \wedge x_3 := 2\mathcal{X}, \\ \top \end{array} \right\}$$

Where, for instance, the element $\mathcal{X} < 4 \wedge x_2 := \mathcal{X} + 1$ is interpreted as x after line two can hold a value in the set of integers less than five.

Besides Π and D the result produced by SA or CSA also depends on how the data-flow equations presented in Definitions 2.2.1 and 2.3.1 respectively have been computed. The most accurate solution is the MOP (Meet Over all Paths) where the least upper bounds of abstract values are taken over all paths leading to a statement and those paths are individually calculated, i.e., the result of transfer functions is never merged but propagated up to that statement. However, the MOP solution for data-flow equations might not always be computable. Thus, the MFP (Maximal Fixed Point) solution is used which is always computable and safely over-approximates the result of MOP [33]. This dissertation focuses on SA and CSA that use the MFP approach for solving their data-flow equations. In the future work the proposed generalized view of analysis will be augmented to consider for MFP and MOP based analyses and analyses that might use mixed MFP and MOP approaches, e.g., refinements in model checking [11] or abstractions in symbolic execution [2].

Thus, in the context of this dissertation a program analysis can be described by two parameters:

1. the precision of the analysis determined by an abstract domain D commonly represented as a lattice L_D; and

2. the set of paths $\Pi \subseteq \Pi_{CFG}$ which the analysis examines.

The only requirement on L_D is that it must relate to the concrete domain L_C of a variable by a Galois connection $(L_C, \alpha, \gamma, L_D)$ such that $\alpha : L_C \rightarrow L_D, \gamma : L_D \rightarrow L_C$ and $\forall l_C \in L_C, \forall l_D \in L_D$ the following correspondence holds: $\alpha(l_C) \preceq l_D$ iff $l_C \preceq \gamma(l_D)$ [33]. This requirement states that there should exist two functions one from the concrete domain to the abstract, called the abstraction function α, and another from the abstract domain to the concrete, called the concretization function γ. Moreover, if the element of the abstract domain over-approximates the abstract value to which an element of the concrete domain is mapped to by the abstraction function then the concretization function must map that abstract element to a concrete value that over-approximates the original concrete value. For example for the domain D_{zero} the abstraction function maps the element of concrete integer domain $\{1\}$ to the $!0$ element of D_{zero}. In turn the concretization function maps the $!0$ to all sets of $\mathcal{P}(\mathbb{Z})$ that do not contain 0. Each such set over-approximates the initial $\{1\}$ set.

For the second parameter Π, three cases can be identified with different Π. In the first case, all paths in Π are loop-free and can be paired with exactly one acyclic paths of CFG. In the second case, paths in Π may loop, but the number of iterations of the loops are bounded by $k < \infty$. In this instance the paths are instantiated from CFG by copying k times the nodes of repeated loops. Finally, if Π contains an unbounded number of loop iterations then paths map directly to paths to the CFG structure, but in this case D has additional restrictions. In particular, the domain D of analysis should satisfy the Ascending Chain Condition which ensures the termination of data-flow analyses [33].

Upon termination an analysis provides the final result that decides that $P \models \phi$. Yet, all analyses implicitly contain information on the set of paths, which is defined as $\widetilde{\Pi}$, on which ϕ holds, which is the set of paths they analyze, i.e., $\widetilde{\Pi} = \Pi$.

However, when SA and CSA cannot decide $P \models \phi$ for Π they may decide it for a subset of the analyzed paths i.e., $\widetilde{\Pi} \subseteq \Pi$, e.g., an infeasible branch for which ϕ trivially holds. In cases when SA or CSA cannot decide $P \models \phi$, its $\widetilde{\Pi} = \emptyset$.

The above discussions result in the following two definitions:

Definition 3.1.1 (Partial result)**.** *For a given program P, property ϕ and a set P's paths Π, if an analysis A runs on paths Π and determines that $P \models \phi$ on $\widetilde{\Pi} \neq \emptyset$ and $\widetilde{\Pi} \subset \Pi$ then $\widetilde{\Pi}$ is a partial result of A.*

Definition 3.1.2 (Analysis instance)**.** *Analysis instance A is determined by an abstract domain D_L and the set of paths Π it analyzes.*

Definition 3.1.3 (Final result)**.** *For a program P and property ϕ, the final result of A is the set of paths $\widetilde{\Pi} \subseteq \Pi_{CFG}$ for which $P \models \phi$, i.e., $\widetilde{\Pi}$ is a function of P, ϕ and A.*

Usually the result of DSE is a single path π, i.e., the path that it has executed, for which DSE generates the path conditions. This dissertation considers an extended version of DSE that can provide the final result for a set of paths that have common prefixes with the executed paths. It does this by negating each condition in the path condition and checking the conjunction of the prefix up to the negated condition and the negated condition for satisfiability. If it is unsatisfiable then the alternative branch of the prefix is infeasible and therefore all paths containing the prefix and that branch cannot violate ϕ. The extended DSE can detect if all alternative branches for the executed path are infeasible and the executed path does not violate ϕ. In the worst case, when all alternative branches are feasible, the extended DSE produces the same result as a regular DSE – only verifies the executed

path. In the rest of the dissertation, all references to DSE imply this extended version of DSE.

3.2 The Space of Π and L_D parameters

Given that the introduced analyses can be described by two parameters L and Π, one can imagine a space of analyses varying with L and Π. Figure 3.1 depicts instances of various analyses in this space, with Π varying along the y-axis and L along the x-axis. On y axis π is a single path, Π' is a set of paths and Π_{CFG} all paths in CFG. On x axis label L_c represents the concrete domain, L_s the symbolic abstract domain and L_p the parity abstract domain. The elements of y-axis are partially ordered by the path inclusion relation, i.e., $\pi \in \Pi_{CFG}$ and $\Pi' \in \Pi_{CFG}$. The elements of x-axis are partially ordered by the domain precision, i.e., $L_c \prec L_s \prec L_p$.

Thus, testing T is identified by the concrete domain L_c and a single path π it is capable of analyzing. DSE is defined over a single path and symbolic abstract domain L_s. The conditional parity static analysis CSA$_p$ is instantiated by a set of paths Π' and the parity abstract domain $L_{parity} \equiv L_p$ while traditional parity static analysis SA$_p$ is instantiated by the same L_p and Π_{CFG}. Some other SA is instantiated by all paths and some abstract domain L_a. The *Ideal* point in Figure 3.1 represents the situation when all information about the program is known, i.e., we can verify any program property for any set of paths. This point corresponds to the case when SE is able to explore all of the program's feasible paths and the facts on the exit and the entry of each basic block is expressed with L_s precision for each path. With this information any assertion property of the program can be verified.

As was discussed in the previous section, the result of an analysis is a set of paths $\widetilde{\Pi} \subset \Pi$ for which ϕ holds. When analysis outputs $\widetilde{\Pi} = \emptyset$ then the analysis cannot provide

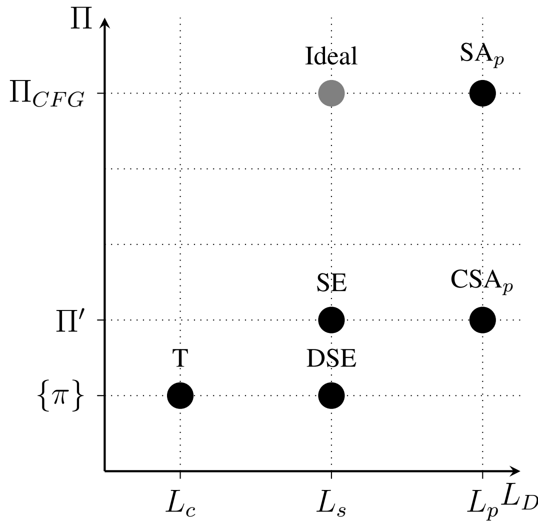


Figure 3.1: Different analysis instantiations based on Π and L.

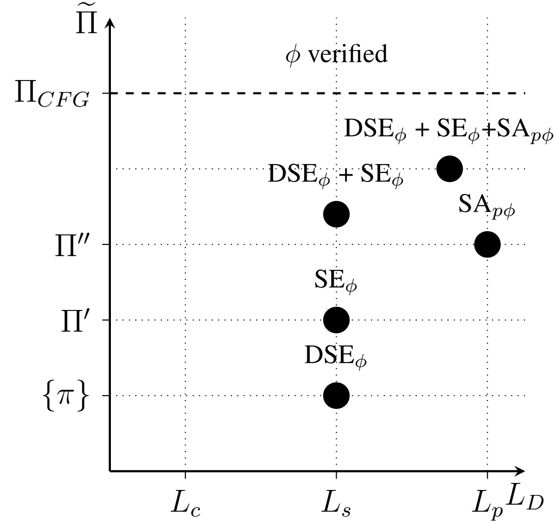


Figure 3.2: Concept of combining results of different analyses.

any definitive information about property satisfaction. When $\widetilde{\Pi} \subset \Pi$ then the analysis produces a partial result. When $\widetilde{\Pi} = \Pi_{CFG}$ then the analysis has determined that the property holds.

Since SE already uses L_s, which is sufficient to give a definite answer on every analyzed path, then $\widetilde{\Pi} = \Pi$ as shown on Figure 3.2. The subscript ϕ identifies the result of an analysis for that property. The extended DSE can have $|\widetilde{\Pi}| \geq 1$ such that $\pi \in \widetilde{\Pi}$. The picture depicts the worst case result for DSE, when $\widetilde{\Pi} = \{\pi\}$. For illustration purposes assume that paths Π, e.g., Π' and Π'' on this Figure, are disjointed. The ultimate objective of combining final results of different analyses is to demonstrate that ϕ holds for all program paths without regard to the values of L_D. Pictorially this equates to the case when the sum of analyses crosses the dashed line on Figure 3.2 – which represents the set of all possible program paths.

The progression of information accumulated when combining the results of multiple analyses together can be seen on Figure 3.2 when the results of SE_ϕ and DSE$_\phi$ are com-

bined. In this case the set of paths for which ϕ holds has increased to $\Pi' \cup \{\pi\}$ which were calculated with the domain precision of L_s. When the information computed by $\text{SA}_{p\phi}$ is added the combined result has moved even closer to the goal line while the value of L shifted in between L_s and L_p, i.e., some paths have information with accuracy of L_s and some with L_p. The next section describes this conceptual representation of combining analyses framework in greater details.

3.3 The Unification Framework Overview

The above analyses parametrization presents the opportunity for controlling analyses through Π and combing their final results $\widetilde{\Pi}$. This section proposes the unification framework that accumulates $\widetilde{\Pi}$ which is referred to as path-property coverage (PPC) and exploits cumulative PPC to control analyses by generating new conditions θ that are translated to Π for the subsequent analyses. Figure 3.3 sketches the unification framework where existing analyses can be adapted to allow them to be integrated into the framework and where PPC of each analysis can be stored and queried. For a correctness property, expressed as an assertion ϕ, and a program P, there are many different static and dynamic analysis techniques $(A_1, \ldots, A_n)$ that might be applied to provide information to developers about the satisfaction or falsification of ϕ.

Early in development, or early in a maintenance phase of P, an analysis A_i, may be able to quickly detect violations of ϕ. In such situations, developers should use A_i as is, but once it becomes difficult to detect violations we advocate a shift in the approach. Existing analyses are modified to add a *reporting* capability, $A_i \circ report_i$, to characterize the set of program executions $\widetilde{\Pi}$ on which A_i can definitively show that ϕ holds. This reporting capability can be thought of as a meta-analysis that runs alongside A_i, but does not modify the operation of the analysis – importantly $A_i \circ report_i$ does *not* attempt to verify ϕ on P.

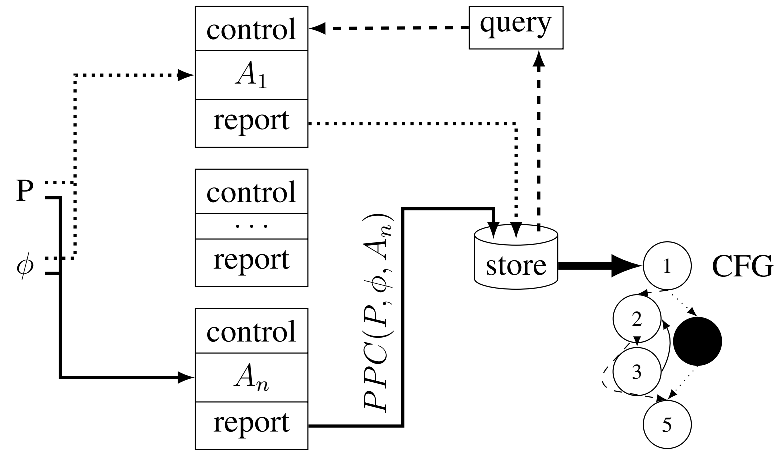


Figure 3.3: Generating and Exploiting PPC

As will be discussed in Chapter 4 a PPC report, $PPC(P, \phi, A_i)$, is a symbolic encoding of a set of program executions Π of P. For example, the set of executions that traverse nodes in the set $\{1, 2, 3, 5\}$ in the CFG could be encoded as the regular language $1; 2; (3; 2)^*; 5$, where ';' denotes concatenation, or as a logical expression that defines the conditions under which the left branch of 1 is taken.

Each analysis run adds potentially new information to a *store* that accumulates PPC for P and ϕ; additional stores can accumulate PPC for other properties. As it will be discussed in Chapter 4, the store is not simply a database of coverage reports, but an integrated representation of the set of paths that have been covered, relative to ϕ, by previous analysis runs. Moreover, it is encoded in a canonical form that can be mapped to representations commonly used by program analyses (such as CFGs), and that can be queried.

One potential use of the store is to render depictions of the extent to which a program's executions have been analyzed and found to conform to ϕ. In the extreme case, such a depiction would simply be a statement that ϕ holds on all executions of P – ϕ is verified. In general, even the application of a series of sophisticated analyses may be unable to achieve such verification. The bottom right of Figure 3.3 sketches a simple visualization of a program's CFG, where *dotted* edges and *filled* nodes indicate that all executions of

those nodes and edges have been shown to conform to ϕ, *dashed* edges indicate that some executions conform, and *solid* edges have not yet been analyzed. Such visualizations could help developers understand the relative effectiveness of different analyses or to decide that a sufficient degree of coverage has been achieved based on their deployment scenario.

Another way to use accumulated PPC is to use it to *control* the application of subsequent analyses but generating the set of path Π for subsequent analyses to consider. For example, just as the dotted edges and solid nodes of the CFG visualization communicate to a developer that no more analysis is needed of those edges and nodes, so too can that information be *queried* from the store and used to restrict the application of an analysis to a portion of the program. Such control information must, of course, be expressed in a form that can be used effectively by a given analysis. For example, a flow-sensitive static analysis might consume a description of the fragment of the CFG over which the analysis should be performed, whereas a symbolic execution or random test generator might use a logical characterization of an input subdomain. In general, with both control and reporting capabilities added to an analysis the combination $control_i \circ A_i \circ report_i$ provides a feedback mechanism to drive the further accumulation of PPC .

The vision sketched in Figure 3.3 is broad in scope and presents numerous research and engineering challenges. The dissertation focuses on defining the foundational concepts that will allow to pursue this vision. Specifically, the next Chapter formalizes PPC and defines how path information related to PPC can be efficiently stored. Section 5.2 illustrates how different analyses can be adapted to report PPC and to allow their application to be controlled by information queried from the PPC store.

Chapter 4

Path Property Coverage

The dissertation focuses on characterizing sets of program executions that conform to explicitly stated correctness properties. This chapter presents a model of sets of program executions and explains how that model can be mapped onto an efficient data structure that permits sets of executions produced from different program analyses to be combined and queried.

4.1 Properties and Paths

This dissertation considers properties that can be encoded as boolean valued expressions embedded into the program, e.g., assertions. To simplify the presentation, only a single such property, ϕ, is considered, but the approach can be generalized to sets of safety properties. This is left for the future work.

As was explained in the previous chapter, the CFG is used as a foundational representation for program executions. The presentation here assumes a single CFG, but interprocedural control flow can be treated in a similar fashion – this is left to future work.

In general, a path in a CFG represents a set of program executions, i.e., those that traverse the path but compute different values. Let p be the location at which ϕ is evaluated. It might happen that after executing the statement at location p some executions end up in a state satisfying ϕ, and in other executions the state at p satisfies $\neg\phi$. Without loss of generality, ϕ can be incorporated into the program as an additional branch condition b_ϕ after the statement p. With this modification a single path through the CFG is able to precisely describe the satisfaction or falsification of ϕ for each program execution.

To formalize, let B be the set of branches in a CFG where each vertex associated with a branch statement is identified by a unique id. Let, as before Π_{CFG} be the set of paths in the CFG; $\hat{\Pi} \subseteq \Pi_{CFG}$ denotes the feasible program paths. A path $\pi \in \Pi_{CFG}$ is uniquely represented by the sequence of the CFG edges that correspond to branch outcomes. Such

an edge is represented as a boolean valued variable b_{id} whose value corresponds to the branch outcome, e.g., b_q is the true outcome of branch q and $\neg b_q$ the false outcome. The value of the ith branch on the path is written $\pi[i]$. It is said that π satisfies ϕ, $\pi \models \phi$, if $\forall i : \pi[i] \neq \neg b_\phi$.

In the unification framework an existing analysis A reports set of paths $\widetilde{\Pi}_A \subseteq \Pi_{CFG}$ for which ϕ holds. Before path property coverage is defined, the concept of covered paths for A is formalized first.

Definition 4.1.1 (Covered paths of an analysis)**.** *Given a program P and correctness property ϕ at the location p then an analysis A covers paths $\widetilde{\Pi}_A \subseteq \Pi_{CFG}$ if*

$$\forall \pi \in \widetilde{\Pi}_A : \pi \models \phi$$

Section 5.2 explains how existing analysis, A, can be adapted to report $\widetilde{\Pi}_A$. Any path that is not covered is referred to as *uncovered*. In general, $\widetilde{\Pi}_A \subset \Pi_{CFG}$ and for particularly buggy programs, or imprecise analyses, it is even possible that $\widetilde{\Pi}_A = \emptyset$, i.e., the analysis is unable to cover any paths.

Using the preceding formal description of Π_A path property coverage is defined as follows

Definition 4.1.2 (Path Property Coverage)**.** *Given a program P with feasible execution paths $\hat{\Pi}$ and correctness property ϕ at the location p, it is said that path property coverage for ϕ is achieved by a set of analyses $\mathbb{A}$ if*

$$\hat{\Pi} \subseteq \bigcup_{A \in \mathbb{A}} \widetilde{\Pi}_A$$

This approach seeks to accumulate subsets of $\hat{\Pi}$ from dynamic analyses and combine them with subsets of Π from static analyses. When analyses calculate that sets of infeasible

paths, i.e., subsets of $\Pi_{CFG} - \hat{\Pi}$, satisfy ϕ they can also be accumulated - since overestimating $\hat{\Pi}$ still permits judgement of PPC.

Unlike traditional test adequacy criteria, these numeric estimates of the percentage of $\hat{\Pi}$ that has been covered by an analysis is not useful or feasible. Such an estimation would require the ability to measure $|\hat{\Pi}|$ and $|\widetilde{\Pi}_A|$, which is not particularly feasible, but the goal of the dissertation is to represent those sets symbolically and to avoid enumerating them.

4.2 Defining Sets of Paths

An analysis need not reason about individual paths. For example, a static data-flow analysis generally reasons about the behavior of the set of execution prefixes (suffixes) that reach each program statement. For instance, it might reason about the set of paths that have different prefixes and suffixes, but that share a common edge $\neg b_q$ in the CFG, i.e., $\{\pi | \exists i : \pi[i] = \neg b_q\}$.

One can represent a set of CFG paths as a regular language $\mathcal{L}$ over an alphabet of branch outcomes.

Definition 4.2.1 (Path Language)**.** *Given a CFG, G, with branch statements, B, the set of CFG paths covered by analysis A, $\mathcal{L}_A$, can be defined as a regular language over $\Sigma = B \cup \bar{B}$, where $\bar{B} = \{\neg b | b \in B\}$. Thus Σ describes false and true outcomes of branch statements.*

A single run of a program that satisfies ϕ, perhaps the simplest form of dynamic analysis, would produce a path language that corresponds to the sequence of branch outcomes executed. If all paths that share the common edge $\neg b_q$ satisfy ϕ, then this is captured by the path language $\mathcal{L}_{\neg b_q} = .^*; \neg b_q; .^*$. A static analysis that demonstrates that all paths in the CFG satisfy ϕ would produce a path language, $\mathcal{L}_{CFG}$, defined by the NFA produced by la-

beling all CFG branch edges with the appropriate element of Σ, labeling all non-branching edges with ϵ, making branch b_ϕ the accept state, and the first statement the start state. In the case of loops there will be repeated occurrences of edges with a given branch id and outcome.

This means of representing a set of paths allows for over-approximating the set of feasible program paths – by including infeasible branch outcomes – which is a common approach to mitigating static analysis cost. Moreover, a path language may include not only infeasible program paths, but the paths that are not even present in $\mathcal{L}_{CFG}$. For instance, a flow-insensitive analysis that is able to prove a property holds would produce a path language Σ^* which significantly overapproximates the $\mathcal{L}_{CFG}$.

A flow-sensitive analysis takes a more refined approach by considering only the branch sequences that correspond to paths in the CFG. Such an analysis can produce a set of paths encoded as a subset of $\mathcal{L}_{CFG}$. For example, $\mathcal{L} = b_q; .^*; \neg b_k$ represents the set of paths starting with b_q taken, followed by some edges and ending with b_k not taken. The path language can then be formed by taking $\mathcal{L} \cap \mathcal{L}_{CFG}$; flow-sensitivity can be enforced after the fact in this way to any path language representation.

Since regular languages are closed under union, the path languages produced by multiple analyses can be combined. Using regular languages, however, may not yield a particularly efficient representation, so path languages are mapped to another encoding where such operation is more efficient.

4.3 Encoding Path Languages

First consider the simplified setting of programs without loops,i.e., CFG paths are a sequence of k elements from Σ without repetition, where $k \leq |B|$. Once the basic framework

for encoding path languages in this setting is presented, the framework will be extended to programs with loops.

Let $\mathbb{B}$ denote the Boolean domain $\{true, false\}$. Partitioning paths into covered, and uncovered, sets is equivalent to defining a Boolean-valued function $\beta_\phi : \Pi_{CFG} \to \mathbb{B}$, which maps π to $true$ if π is covered and to $false$ otherwise. Since the primary interest is in universal properties, i.e., properties that hold on all feasible program executions, then it is implied that an infeasible path trivially satisfies any property ϕ. Because of this, the function β_ϕ does not distinguish feasible and infeasible paths.

Any $\pi \in \Pi_{CFG}$ can be expressed as a Boolean function $\gamma_\pi : \mathbb{B}^k \to \mathbb{B}$. Thus, β_ϕ can be expressed as a disjunction of such functions, $\beta_\phi = \bigvee_{\pi \in \Pi_{cov}} \gamma_\pi$. This reduces the problem of combining covered paths to logical operations on k boolean variables and the problem of finding path property coverage for ϕ to determining when β_ϕ becomes a tautology, i.e., $\beta_\phi \equiv 1$.

For this purpose a bijection is established between the set of branches B and a set of boolean variables V, i.e., $v : B \to V$. Then γ_π can be defined as a function $\gamma_\pi(v_1, \ldots, v_i, \ldots, v_k)$, where $v_i \in V$, such that γ_π is true if the assignments to v_i correspond to the sequence of branches taken in π. The assignments corresponding to π are defined as follows. If $\exists b_i \in \pi$ then $v_i = true$, if $\exists \neg b_i \in \pi$ then $v_i = false$. If none of two cases apply, i.e., b_i is not executed in π, then value returned by γ_π does not depend on v_i.

One approach to building γ_π is to form the disjunction of boolean expressions corresponding to each k-tuple that is consistent with π. A boolean expression for a k-tuple is built as the conjunction of literals l_i, which may be either v_i and $\neg v_i$. Note that when l_j does not appear in π, then the two k-tuples $(l_1, \ldots, true, \ldots, l_k)$ and $(l_1, \ldots, false, \ldots, l_k)$, where $true$ and $false$ are in the ith position, must be disjoined:

$$(l_1 \wedge \cdots \wedge v_j \wedge \cdots \wedge l_k) \vee (l_1 \wedge \cdots \wedge \neg v_j \wedge \cdots \wedge l_k)$$

which simplifies to

$$(l_1 \wedge \cdots \wedge l_{i-1} \wedge l_{i+1} \wedge \cdots \wedge l_k)$$

Therefore each γ_π can be expressed as the conjunction of the literals for branch variables present in π while leaving literals for variables like v_j out.

This simplification allows to define γ not only for a single path but also for a set of paths expressed as a path language, $\gamma_\mathcal{L}$. Sequences of wildcard characters $.^*$ have a similar semantics as the unexecuted branch variable v_j discussed above. That is $\gamma_\mathcal{L}$ evaluates to $true$ when the values of variables appearing in the regular expression are set to their tuple's value in that expression and the assignments to the rest of variables does not affect the result.

For example, the regular expression $.^*; b_q; .^*$ encoding the result of a flow insensitive analysis would be encoded as $\gamma_1 = v_q$. The encoding of the flow sensitive analysis' covered paths $b_q; .^*; \neg b_k$ would be translated into $\gamma_2 = v_q \wedge \neg v_k$. If these analyses target the same ϕ, then their coverage can be combined as $\beta_\phi = \gamma_1 \vee \gamma_2 = v_q \vee (v_q \wedge \neg v_k)$.

4.3.1 BDDs for PPC

Instead of expressing β_ϕ in this Boolean logic format the dissertation utilizes the more efficient Reduced Ordered Binary Decision Diagram (BDD) encoding [9]. A BDD is a rooted, acyclic, directed graph with only two leaf nodes 1 ($true$) and 0 ($false$). Figure 4.1 illustrates several BDDs. Intermediate nodes are called decision nodes and correspond to boolean variables. Each decision node has two children: *low*, when its variable evaluates to 0 (shown as a dashed edge), and *high*, when its variable is evaluated to 1 (shown as a solid edge). The path from the root of the tree to the (shaded) leaf node 1 corresponds to the assignment of boolean variables for which β_ϕ evaluates to $true$. The upper left BDD

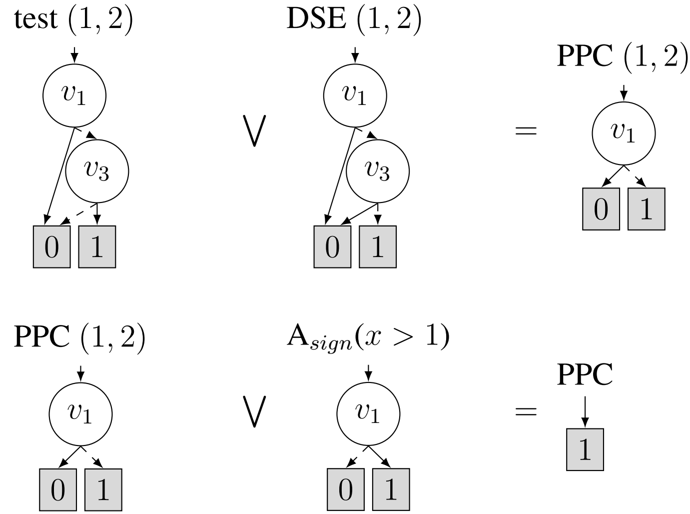


Figure 4.1: PPC store BDD encodings for Example

in Figure 4.1 evaluates to true when $\neg v_1 \wedge v_3$. When $\beta_\phi \equiv 1$ the BDD encoding is a single leaf node 1; see the lower right BDD in Figure 4.1.

Definition 4.3.1 (PPC Store)**.** *Given a path language, $\mathcal{L}$, defined over an alphabet of branch literals, $B \cup \bar{B}$, representing a set of covered paths, $store(\mathcal{L})$ is a BDD defined over a set of variables $V = \{v(b) | b \in B\}$ according to the rules:*

$$store(\epsilon) = false,$$
$$store(.) = true,$$
$$store(b) = v(b),$$
$$store(\neg b) = \neg v(b),$$
$$store(e_1; e_2) = store(e_1) \wedge store(e_2),$$
$$store(e_1 | e_2) = store(e_1) \vee store(e_2)$$

An analysis technique exploits the information it calculates to encode the most general formulation of its path language and associated PPC store. To illustrate, consider the PPC store computed for the example presented in Figure 1.3. When test $(1,2)$ is run then the

false branch of condition $(x > 1)$ is taken – this branch is represented by v_1 in the BDD – and then the assert condition is true – the assert branch is represented by v_3. Hence the BDD labeled "test $(1, 2)$" is generated which represents $\neg v_1 \wedge v_3$. The dynamic symbolic execution of that path is able to prove that the false branch for the assert condition is infeasible so that path, encoded by the BDD labeled "DSE $(1, 2)$", is accumulated in the PPC store, "PPC $(1, 2)$". After the sign analysis A_{sign} is run on the sub-CFG associated with $x > 1$, it is able to determine that all paths through the loop result in the false branch for the assert being infeasible. When the "$\mathrm{A}_{sign}(x > 1)$" BDD is added to the PPC store the result is the value 1 which indicates that path property coverage has been achieved. Section 4.3.3 provides additional examples of BBDs for PPC.

Using a BDD for encoding β_ϕ leads to a compact encoding of path property coverage. The efficiency of the BDD encoding strongly depends on the variable ordering [9]. A variable order that corresponds to the order in which branches appear in the program's text is chosen for the experiments for the sake of convenience. The future work will explore the efficiencies of different orderings.

4.3.2 Iterative Paths

Some analyses explicitly analyze individual loop iterations. For example, a dynamic analysis analyzes each iteration as it executes the program and a symbolic execution considers a bounded set of unrolled loop iterations. For such analyses, a generated path language may contain multiple occurrences of a branch b_i occurring within a loop. To consider this case $store(\mathcal{L})$ is adapted to accommodate analysis results for iterative programs by producing unique branch identifiers for branches within loops. Specifically, for a branch b_i in loop l, the framework creates $b_{i_1}, \ldots, b_{i_{max(l)}}$ – one for each loop iteration. This branch naming

scheme is generalized appropriately for nested loops, i.e., for a loop nest $l_1, \ldots, l_k$ a branch in the innermost loop will have $\prod_{i=1\ldots k} max(l_i)$ unique branch identifiers.

Analyses may explore different numbers of loop iterations and this may cause the domain of β_ϕ to increase over time. The creation of new BDD variables does not invalidate the previously recorded BDD results since any new variables were, by definition, not involved in previous analyses.

It is important to note, that many static analyses, such as data flow analyses, need not unroll loops. When they are able to prove that a property holds they do so by reasoning about all paths through a region of the program. In this case, the specific branching within any loops in the region is, by virtue of the analysis result, irrelevant to property satisfaction and those branches can be left out of the path language and associated BDD encoding. This is precisely why the sign analysis A_{sign}, when applied to $x > 1$, did not produce a BDD which includes variables modeling the branch condition for the `while` loop. The abstract values flowing out of the loop, $(+, +)$, provided all of the information necessary about the inner workings of the loop to determine property satisfaction.

4.3.3 Assessing and Querying PPC

The path adequacy criterion is based on "a priori" knowledge of the total number of feasible paths in a program. The difficulty of determining that number using static code analyses makes assessing the percentage of path coverage intractable due to the presence of infeasible paths and potentially infinitely many paths.

The dissertation takes an alternative approach, which abstracts from path coverage and considers *execution subtree coverage*. This approach does not require the explicit construction of execution trees.To illustrate the concept of subtree coverage and how to obtain the subtree coverage information, consider the illustration in Figure 4.2.

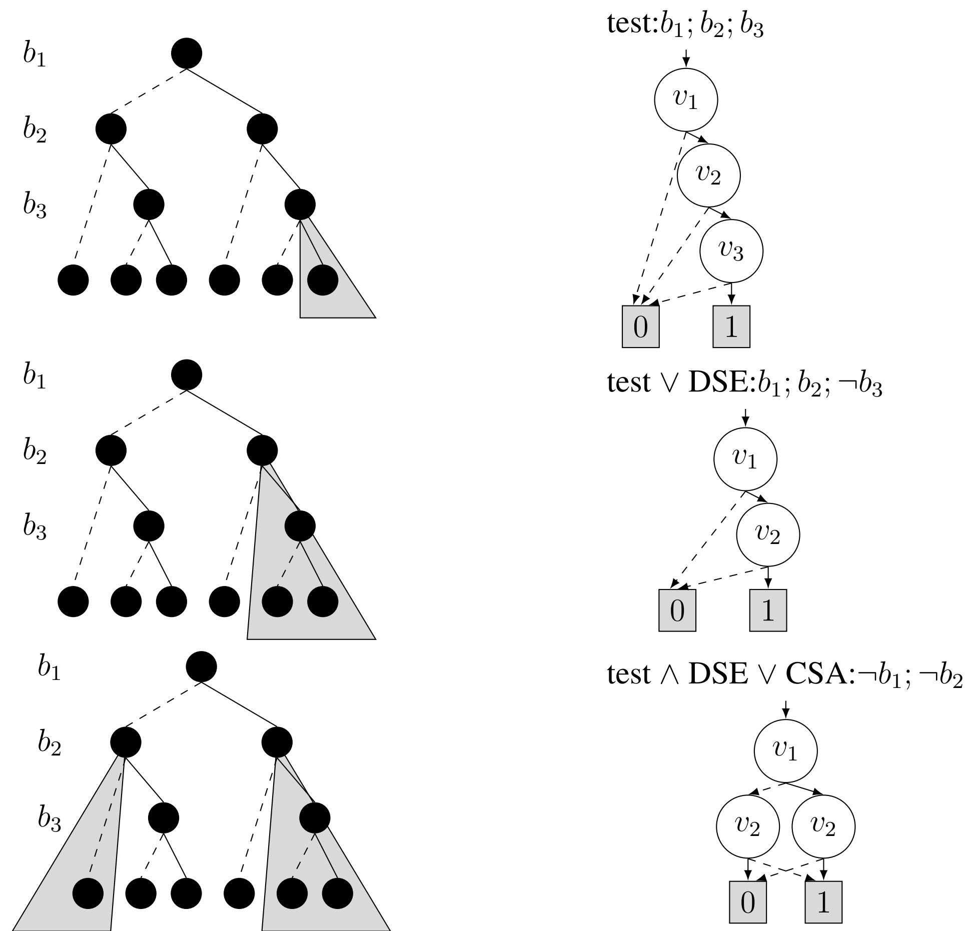


Figure 4.2: Explanation of subtree coverage criterion

Each of the three figures corresponds to information associated with the set of paths covered by a test run (top), the prior test run and DSE (middle), and the previous test and DSE runs and conditional static analysis (CSA) (bottom). The left side of each figure depicts the execution tree of a program with their conditional statements b_i labels on the left. The covered subtrees of the execution tree as the result of the corresponding analyses run are shaded in grey. The right side depicts the PPC store's BDD representation of analyses result.

In the top picture a test run executes a single path. This results in the coverage of a single subtree, on b_3 level, that consists of a single leaf node indicating that the true branch

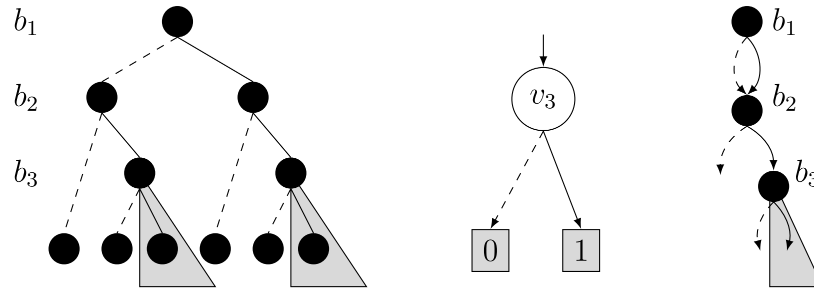


Figure 4.3: Special case of sub-tree coverage for SA:b_3: execution tree(left), BDD(middle), CFG(right)

b_3 has been covered in the context of $b_1 \wedge b_2$ prefix. The right side of the figure shows this information encoded in BDD form.

The middle picture is more interesting. It shows that when the executed path is analyzed by DSE and $\neg b_3$ is found to be infeasible, then a larger subtree is covered. This means that after executing the true branch of b_1 and then the true branch of b_2, all subsequent program behavior is guaranteed to be consistent with ϕ – either because it was observed directly or it was inferred by DSE. It is said that the *prefix length* of this covered sub-tree is 2. The BDD on the right depicts the PPC store where it can be seen that the prefix length is nothing more than the length of a path from the root of the BDD to 1.

The bottom picture shows the scenario when an additional static analysis determines that $\neg b_1, \neg b_2$ is infeasible. As a result another subtree on level 3 has been covered. The two covered subtrees are described by the two paths of length 2 of the corresponding BDD. (Note, that $\neg b_1$ might be infeasible by itself because SA over-approximates the behavior.)

When execution tree subtrees corresponding to a branch of a conditional statement b_i have been covered, the BDD will collapse those prefixes and have that branch only at the top of BDD. This situation is shown on Figure 4.3 when SA has determined that all paths that contain the true branch of b_3 do not violate ϕ. In this case the information from the BDD can be mapped straight to the CFG which indicates that all subtrees in the execution tree of the true branch of b_3 have been evaluated. Thus all CFG nodes which are reachable

from the CFG entry node through the true branch of b_3 cannot contribute to ϕ violation – the CFG coverage is depicted on the right of Figurefig:SepcialTreeCoverage.

Since testers are more familiar with CFG than with execution trees, the subtree coverage of CFG as in the last example should be more intuitive than the subtree coverage of execution trees shown on Figure 4.2. Yet, these cases can also be projected to the CFG by introducing some notion of "partially" covered subtrees. This degree of partially cannot be obtained from a BDD directly and more likely should be used together with CFG. This effort is left for future work. The following definition of the proposed subtree coverage is defined not for a set of test as regular adequacy criteria but for the set of program analyses.

Definition 4.3.2 (Subtree adequacy criterion)**.** *The analysis set $\mathbb{A}$ is l-level subtree adequate for a program P, if for every subtree with prefix of size l in P's CFG, there exist some subset of analyses $\mathbb{A}' \subseteq \mathbb{A}$ that covers that subtree. $\mathbb{A}$ is adequate when all subtrees at 0 level are covered, i.e., when the corresponding BDD returns $true$ value.*

Note, that no additional data structures besides BDD are needed to obtain the subtree coverage information.

In general, the previous examples illustrate that the smaller the prefix lengths for covered subtrees in the BDD the *greater coverage* of paths. The intuition is that the lower the level of subtrees that are covered, the closer the program is to being PPC adequate. Characterizing the length and number of prefixes in the PPC store provides insight into how an analysis contributes new PPC coverage information.

In addition to characterizing coverage information, the BDD can be used to extract the information about uncovered program paths that should be targeted by future analyses. All paths in a BDD that lead to 0 leaf node describe the set of uncovered program paths. For example at the top picture of Figure 4.2 there are three paths $\neg v_1$, $v_1; \neg v_2$ and $v_1; v_2; \neg v_3$

leading to 0 leaf node. Thus the paths with prefixes b_1, $b_1; \neg b_2$ and $b_1; b_2; \neg b_3$ should be analyzed by the next analyses.

The translation of a prefix to θ condition of CSA is straightforward. It would be valuable for the unification framework to keep track of θ for each CSA to avoid the repetitive analysis runs that will increase PPC in BDD store.

DSE can execute a given prefix if it is expressed as pc. The translation from PPC encoding to pc can be done inside the unification framework provided the framework keeps pc for previously executed DSE or SE runs. When the unanalyzed prefix is given, the unification framework can look at the analyzed DSE and SE paths and use pc of the path with the longest common prefix up to the last condition and the last condition is the negation of the prefix's last condition to generate input values that will explore the needed prefix. For example, if prefix is $b_1; b_2; \neg b_3$ then the framework chooses a path containing $b_1; b_2; b_3$. Next it selects the first three conditions of the corresponding pc which are $pc_{b_1} \wedge pc_{b_2} \wedge pc_{b_3}$. After that the last condition is negated to produce $pc' = pc_{b_1} \wedge pc_{b_2} \wedge \neg pc_{b_3}$. Solving pc' will provide DSE with an input to traverse the needed prefix.

The next Chapter explains how the unification framework has been instantiated and how the existing analyses have been adapted to fit into the unification framework.

Chapter 5

Instantiation of the Unification Framework

All implementations described in this chapter are written in the Java programming language. The first part of this chapter describes the instantiation of the unification framework depicted on Figure 3.3 while the second part presents the implementations of CSA and DSE and their adaptations which allow for the analyses integration into the unification framework. The presentation of each implementation starts with an algorithm description followed by discussion of essential implementation details.

5.1 Unification Framework

The unification framework has two main functions. One function is the accumulation of PPC. This task is accomplished by obtaining PPC information from a set of analyses and saving it in the PPC store. The second objective is the generation of analysis control. This is done by querying the PPC store for unexplored paths and passing that information in the form of control to the next analysis in the execution queue. As discussed in the previous chapter, this framework uses a BDD as its choice of the PPC store. The language $\mathcal{L}$ for a set of paths of PPC is encoded as a set of branch literals, called a $prefix$.

5.1.1 Algorithm

Algorithm 5.1 describes the implementation of the unification framework presented in Chapter 3. The algorithm consists of three functions: $UnificationFramework$ which is the core algorithm, and the two auxiliary functions $generate$ and $accumulate$ that correspond to the framework's two main functionalities.

The $UnificationFramework$ receives two parameters: a program P and b_ϕ which is the conditional statement of the assertion ϕ. When the false branch of b_ϕ, i.e., $\neg l_\phi$ (recall that l_q is a branch literal for b_q) is executed the assertion is violated. On line 2 the framework

Algorithm 5.1 Unification framework instantiation

```
 1: function UnificationFramework(P,b_φ)
 2:     𝔸 ← {A_1, A_2, ..., A_n}
 3:     bddStore ← false
 4:     while bddStore ≠ true and ¬resources do
 5:         A ← select(𝔸)
 6:         Π ← generate(bddStore)
 7:         PPC ← A(P, b_φ, Π)
 8:         bddStore ← accumulate(bddStore, PPC)
 9:     end while
10:     report(bddStore)
11: end function
12: function generate(bddStore)
13:     for all unsatCube ∈ bddStore do
14:         prefix ← convertToPPC(unsatCube)
15:         Π ← Π ∪ {prefix}
16:     end for
17:     return Π
18: end function
19: function accumulate(bddStore, PPC)
20:     for all prefix ∈ PPC do
21:         cube ← convertToBDD(prefix)
22:         bddStore ← bddStore ∨ cube
23:     end for
24:     return bddStore
25: end function
```

initializes the set of analyses $\mathbb{A}$ that later it can dispatch. Line 3 initializes $bddStore$, the BDD implementation of the PPC store, to $false$, i.e., an empty set of analyzed paths.

The algorithm next iterates lines 4 through 9 until either all paths have been analyzed, i.e., $bddStore$ is $true$, or the resources threshold has been reached. These two conditions guarantee the termination of the algorithm. If the algorithm terminates when $bddStore == true$ then it reports on line 10 that $P \models \phi$. If the termination occurs because of the resources condition has been satisfied then the algorithm reports the condition for which $P \models \phi$.

On line 5 the algorithm selects the next analysis A to be executed. Line 6 generates a set of prefixes that describes the set of uncovered paths A must analyze. On line 7 the framework runs the analysis and assigns its result, i.e., the set of prefixes, to the PPC variable. Line 8 adds the analysis result into the PPC store.

The $generate$ function on lines 12-18 takes the BDD store as an argument. It queries the BDD for unsatisfiable cubes. An unsatisfiable cube is a set of BDD variables of a path leading to the BDD's $zero$ node. On line 14 each such cube is converted to the corresponding set of branch literals, i.e., a prefix. The resulting prefix is added to the set Π is returned to the main algorithm.

The $accumulate$ function takes as its input the BDD store and the set of prefixes PPC. Then on line 21 each prefix is translated to its BDD's representation, i.e., a cube – a satisfiable assignment. Then the cube is added to the BDD store using the disjunction operator.

The set of analysis $\mathbb{A}$ can contain a special analysis A_{slice} that computes that static forward slice of the program variables involved in the property ϕ; this is similar to [27]. Any paths that lie outside the slice trivially satisfy ϕ and thus are reported as covered in PPC. This is called *property based slicing*. If such analysis is present then the framework should be directed to execute A_{slice} first.

5.1.2 Implementation

In the current implementation the set $\mathbb{A}$ consists of three intra-procedural analyses: two CSA with D_{zero} and D_{sign} domains, and DSE. The analysis A_{slice} also has been implemented. The intention of the dissertation is not to implement the most precise or sophisticated analysis possible, but rather to understand how different types of analyses can provide, or fail to provide, coverage of program paths. The next section describes in detail how these three analyses have been implemented.

The PPC store is implemented using JDD [29] which is an open source decision diagram library written in Java. It has a rich API that allowed for querying for satisfying assignments, determining various BDD statistics, and producing visual depictions of the BDDs. To collect PPC information, the set of satisfying assignments for the BDD is analyzed which is used in reporting.

The reporting produces the visual representation of the BDD in `dot` format and also displays the distribution of the lengths of cubes, i.e., the satisfying paths. As discussed in Section 4.3.3 the length and number of the BDD's paths provide insights into how an analysis contributes PPC. This information provides useful data to a researcher and may not be relevant to end user.

5.2 Adapting Existing Analyses

In general, in order to incorporate a new analysis A into the unification framework, A must be extended to have at least the PPC reporting capability and preferably the controlling capability. The effort required to implement these two features for A depends on the internal design of A. As the following sections show extending the reporting capability for DSE is straightforward because it only requires the tracking of the executed path. However, adding reporting to CSA requires additional post processing of the calculated intermediate result. Yet, implementing the control feature for CSA requires less effort than implementing it for DSE, since controlling DSE requires the translation from PPC to the concrete input values. Nevertheless, A is only extended once but that effort allows the final results of A to be combined with any existing analysis in $\mathbb{A}$.

This section discusses two different analyses and describes their adapted algorithms that permit the analyses to be integrated into the unification framework. Specifically, it demonstrates how a conditional static data flow analysis and the execution of a dynamic

symbolic execution can be modified to generate PPC and parameterized to target a portion of a program's input domain.

5.2.1 Conditional Data-Flow Analysis

For PPC, only the set of paths that do not violate ϕ are considered. For an analysis to determine that no paths violate the assertion, it must be the case that the false branch b_ϕ is infeasible. If such a judgement cannot be made, then any branch outcomes that can be judged to be infeasible will contribute to PPC. If an infeasible branch outcome is found, then all paths that the data-flow analysis traversed leading up to that branch outcome are guaranteed to not violate ϕ. Thus the goal of CSA is to determine infeasible branches given the condition θ.

In this section the algorithm for CSA is presented in two steps. First the core data-flow algorithm is described that computes entry and exit facts for each statement, i.e., the intermediate facts. The second algorithm describes how infeasible branches are determined from those facts.

5.2.1.1 Algorithm

There are several algorithms that compute data flow equations described in Definition 2.2.1. Those algorithms compute a fix point of the intermediate facts, i.e., when upon iteration over statements no changes to the intermediate facts are detected. The algorithm described below is a work-list based algorithm similar to the one sketched in [3]. Instead of iterating over all statements, work-list based algorithms only consider those statements for which the incoming facts have been changed.

Algorithm 5.2 describes the core data-flow algorithm of CSA, which computes the intermediate facts.The function $WorkList$ takes three parameters: program P, the set of

Algorithm 5.2 A work-list algorithm for conditional static analysis

```
1: function WorkList(P,θ, L)
2:     ({s}, {(l, l′)}, e) ← CFG(P)
3:     e_in ← ⊤
4:     s_out ← ⊥ ∀s ∈ {s}
5:     W ← {s}
6:     while W ≠ ∅ do
7:         n ← remove a statement from W
8:         oldState ← n_out
9:         n_in ← ∪p_out ∀p ∈ predecessors(n)
10:        if n is a conditional statement then
11:            (l, l′)_t ← true branch of n
12:            (l, l′)_f ← false branch of n
13:            if (l, l′)_t ∈ θ then
14:                n_out_t ← ⊥
15:                n_out_f ← F_n_f(n_in)
16:            else if (l, l′)_f ∈ θ then
17:                n_out_t ← F_n_t(n_in)
18:                n_out_f ← ⊥
19:            else
20:                n_out_t ← F_n_t(n_in)
21:                n_out_f ← F_n_f(n_in)
22:            end if
23:        else
24:            n_out ← F_n(n_in)
25:        end if
26:        if oldSate ≠ n_out then
27:            for all c ∈ successor(n) do
28:                if c ∉ W then
29:                    W ← W ∪ {c}
30:                end if
31:            end for
32:        end if
33:    end while
34: end function
```

excluded edges θ and the abstract domain L that defines transfer functions $\mathcal{F}$ used by the analysis.

First the algorithm initializes the three tuple $(\{s\}, \{(l, l')\}, e)$ from the CFG of P, where $\{s\}$ is the set of statements, $\{(l, l')\}$ is the set of edges and e is the entry statement. Next, on

lines 3-5 the algorithm initializes the incoming facts of e to the top element, the outgoing facts of all statement to the bottom element and adds all statements to the work-list W.

Then while the work-list is not empty, a statement is removed from W and processed in the following way. First, on line 8 the statement's current outgoing facts are recorded. This information will be used later to decide if a new statement should be added to W or not. Next, on line 9 all incoming facts from its predecessor are combined. If n is not a conditional statement then on line 24 a transfer function of that statement is applied to the incoming facts of n to compute the outgoing fact of the statement.

A conditional statement receives a special consideration on lines 11 through 21. First, on lines 11 and 12 the true and false branches of n are identified. If one of those branches are in the exclude set θ then the outgoing facts for those branches are set to the bottom element, i.e., lines 14 and 18. Otherwise the branches are processed as usual.

At the end of computation the fresh facts for n_{out} are compared to their old values stored in $oldSate$. If the values are not the same then all successor for n are added to work-list W for processing.

Algorithm 5.3 uses these computed intermediate facts to produce the set of infeasible branches. CSA is parametrized by the program P, the conditional statement of ϕ, the set of exclude edges and the abstract domain of the analysis. The first, third and fourth parameters are passed straight to the core algorithm and b_ϕ is used locally for the detecting infeasible branches.

After invoking the $WorkList$ function on line 4 the algorithm initializes the set of infeasible branches to an empty set and identifies the true branch of b_ϕ. Then the algorithm iterates over all branches of the program to determine infeasible branches.

A branch is infeasible if 1) the value of its incoming fact is not $\perp$, because it means that its predecessors are already infeasible; 2) its outgoing facts have $\perp$ value, i.e., the main criteria for infeasibility; 3) the branch is not part of the exclude set and 4) the branch is not

Algorithm 5.3 Algorithm for conditional static analysis

1: **function** CSA(P,b_ϕ, θ, L)
2: $WorkList(P, \theta, L)$
3: $B \leftarrow conditionalStatements(P)$
4: $infeasibleBranches \leftarrow \emptyset$
5: $(l_\phi, l) \leftarrow$ true branch of b_ϕ
6: **for all** $(l_b, l) \in CFG(P) : b \in B$ **do**
7: **if** $b_{in} \neq \perp \wedge (l_b, l)_{out} == \perp \wedge (l_b, l) \notin \theta \wedge (l_b, l) \neq (l_\phi, l)$ **then**
8: $infeasibleBranches \leftarrow infeasibleBranches \cup \{(l_b, l)\}$
9: **end if**
10: **end for**
11: **return** $infeasibleBranches$
12: **end function**

the true outcome of the conditional statement of ϕ, because the goal of the analyses is to show that $\neg(l_\phi, l)$ is infeasible. Detecting infeasibility of $\neg(l_\phi, l)$ can only raise a warning that under θ, ϕ is violated, but it is unknown whether θ corresponds to a feasible set of paths.

5.2.1.2 Implementation

The core data-flow algorithm described by the $WorkList$ function has been implemented using the Soot [44] Java optimization framework. Soot has a rich set of extensible static analysis components that can be utilized to implement a custom static analysis such as CSA with different abstract domains. In this case the branched forward analysis has been extended to implement the $WorkList$ function.

While this current implementation does cause the analysis to traverse the excluded sub-CFG, no information from that sub-CFG is computed or merged. Another alternative would be to modify the CFG by removing exclude edges to control the analysis.

Algorithm 5.4 Adapting conditional static analysis

```
1: function CSA_i(P,b_φ,Π)
2:     L ← L_i
3:     θ ← excludeBranches(P, Π)
4:     Δ ← CSA(P, b_φ, θ, L)
5:     PPC ← ∅
6:     for all infeasibleBranch ∈ Δ do
7:         {prefix} ← intersect(P, Π, infeasibleBranch)
8:         PPC ← PPC ∪ {prefix}
9:     end for
10:    return PPC
11: end function
```

5.2.2 Adapting a Conditional Data-Flow Analysis

This section describes how the above CSA algorithm can be adapted to fit into the unification framework. To be incorporated in the unification framework the analysis $A \in \mathbb{A}$ should accept the set of paths to analyze, Π, described by a set of prefixes and return PPC which is also a set of prefixes. Thus, the purpose of the adaptation is to augment the main CSA algorithm with two functionalities: translating Π to θ and translating $infeasibleBranches$ to PPC.

5.2.2.1 Algorithm

Algorithm 5.4 describes the steps for extending CSA. CSA is parametrized by a program P, the property's conditional statement b_ϕ and a set of paths Π it must analyze. Line 2 of the algorithm assigns the abstract domain for i^{th} analysis instance. On line 3 the algorithm calls $excludeBranches$ method that translated Π into the set of exclude edges θ. This function uses the CFG of P for identifying those edges.

On line 4 the core CSA algorithm returns the set of infeasible branches Δ expressed through branch literals. The rest of the algorithm describes how each branch literal is combined with θ to produce PPC for the unification framework.

The set of infeasible paths, PPC, is initialized to an empty set on line 5. Next, each infeasible branch is translated into the path language that can be described by a set of prefixes where each prefix describes a distinct path in CFG, i.e., paths that don't subsume each other. This is done on line 7 by invoking the $intersect$ function. This function intersects $\mathcal{L}_{CFG}$, the language of the analyzed paths Π and the language of $infeasibleBranch$. On line 7 the resulting set of prefixes is added to PPC. After processing all infeasible branches the analysis returns PPC.

To illustrate how PPC of CSA is produced according to the algorithm, consider the controlled sign analysis A_{sign} on the right side of Figure 1.4. In this case Π is described by a single prefix $\{b_{x>1}\}$ that is translated to the exclude set $\theta = \{\neg b_{x>1}\}$. This causes the analysis to avoid the sub-CFG rooted at that branch outcome. The resulting analysis is able to determine that $\neg b_{\phi}$, the false outcome of the $x \neq 0$ test, is infeasible – since $x \mapsto +$ is inconsistent with $x \neq 0$. The function $intersect$ takes the language of Π

$$\mathcal{L}_{\Pi} = .^{*}; b_{x>1}; .^{*}$$

and intersects it with the language of the infeasible branch

$$\mathcal{L}_{\neg b_{\phi}} = .^{*}; \neg b_{\phi}; .^{*}$$

resulting in

$$\mathcal{L}_{\Pi \cap \neg b_{\phi}} = .^{*}; b_{x>1}; .^{*}; \neg b_{\phi}$$

Next, the result is intersected with the language of CFG $\mathcal{L}_{CFG}$ giving the final result of

$$\mathcal{L}_{\Pi \cap \neg b_{\phi} \cap CFG} = b_{x>1}; .^{*}; \neg b_{\phi}$$

which is expressed as the set of branch literals $\{b_{x>1}, \neg b_\phi\}$.

5.2.2.2 Implementation

The adaptation of CSA is implemented for CSA$_{zero}$ and CSA$_{sign}$ instances of CSA. They both use the same data-flow implementation only the former is instantiated with D_{zero} domain and the latter with D_{sign} domain as depicted in Figure 2.1. When multiple variables are present, the lattice is simply the product of the per-variable lattices. The analyses can reason about variables of `int` and `double` types.

The same data-flow implementation implies that all reporting and controlling capabilities have been added once and adding another CSA with a domain D to $\mathbb{A}$ reduces to implementing the transfer functions associated with D.

The auxiliary function $excludeBranches$ and $intersect$ are implemented by traversing the CFG with the DFS graph search algorithm. For example, $intersect$ traverses all CFG paths and selects those that are described by Π and contain $infeasibleBranch$.

5.2.3 Adapting Dynamic Symbolic Execution

5.2.3.1 Algorithm

Algorithm 5.5 describes how the DSE analysis is adapted to fit into the unification framework. The algorithm takes a program P, b_ϕ and Π. The b_ϕ value allows DSE to avoid collecting pc after b_ϕ has been executed. Next the set of paths given by the unification framework is translated to the concrete input values that traverse a path from Π. Unlike a conventional DSE that returns a set of pc this version of DSE returns a set of tuples $\Delta = \{(branch, pc)\}$ where $branch$ is the branch literal of an executed branch and pc is the path condition that governs the execution of that branch. The result of DSE run is assigned to Δ on line 3.

Algorithm 5.5 Adapting dynamic symbolic execution

```
1:  function DSE(P,b_φ,Π)
2:      θ ← getInputs(P, Π)
3:      Δ ← DSE(P, φ, θ)
4:      PPC ← ∅
5:      prefix ← ∅
6:      PC ← true
7:      for all (branch, pc) ∈ Δ do
8:          PC′ ← PC ∧ ¬pc
9:          if ¬SAT(PC′) then
10:             prefix′ ← prefix ∪ {¬branch}
11:             PPC ← PPC ∪ {prefix′}
12:         end if
13:         PC ← PC ∧ pc
14:         prefix ← prefix ∪ {branch}
15:     end for
16:     PPC ← PPC ∪ {prefix}
17:     return PPC
18: end function
```

The rest of the algorithm constructs PPC which is initialized to an empty set. As before $prefix$ contains the set of branch literals describing a set of paths. The PC variable stores the conjunctions of path conditions of the current $prefix$. The initial value of PC is set to true on line 6.

Besides constructing $prefix$ for the executed path lines 7 through 15 also attempt to generate a richer PPC using PC. If it can be determined that $\neg pc_i$ is unsatisfiable then it means that all paths with the prefix

$$l_1; \ldots; \neg l_i$$

will not violate ϕ and therefore can be added to PPC. This is what lines 8-12 do. Next, the values of PC and $prefix$ are updated. Upon loop termination the prefix for Π is added to PPC. The algorithm returns generated PPC to the unification framework.

5.2.3.2 Implementation

The current implementation of Algorithm 5.5 does not generate the concrete input values for a given Π. As discussed at the end of the previous chapter, to enable logical representation of Π an additional data structure is required to store Δ of the previous DSE runs. Presently $getInputs$ function returns a random generated input values or input values from a test suite.

Note that the approach of controlling an analysis by restricting its input domain differs from the approach used in static analysis. The latter uses the structure of CFG to effectively restrict its input domain, while the former uses the explicit description of input domain of variables. Some analysis, such as symbolic execution (SE), could use both types of control information. The input domain of SE can be restricted by either providing preconditions on input variables expressed through a set of constraints or during the SE where the symbolic executor can choose to avoid some branch outcomes.

Instead of using an existing symbolic execution framework, such as Symbolic PathFinder (SPF) [36] a new DSE has been implemented. This route simplifies the mapping of static analyses and DSE results to branch outcomes – both analyses express their results in terms of Soot's 3-address intermediate representation, `jimple`.

DSE implementation can be done in one of two ways: instrumenting P to generate pc and instrumenting P to generate traces from which pc are constructed. In the first approach [23, 42, 16] DSE executes a program while simultaneously building pc through instrumentation. The instrumentation consists of calls to a symbolic interpreter that keeps track of symbolic states of each variable. In the second approach [41] a program is instrumented to produce a trace that can be post processed by a symbolic interpreter. The trace-based approach consists of two steps: executing a program while recording a trace and interpreting the trace to generate pc. The implemented DSE uses this trace-based ap-

proach. As mentioned in the algorithm description, besides the information needed for pc generation a program is also instrumented to record executed branches.

Soot is used to instrument subject programs. It allows to inject code at the program expression level to record analysis results and perform dynamic symbolic execution. This instrumentation tracks the outcome and the condition of the taken branch and expressions of variables. Upon program termination the instrumentation accumulated information is recorded in a file. That file is post-processed to produce the sequence of executed branches and path condition generated at each branch.

As discussed in the algorithm each branch outcome of the executed path is analyzed to determine whether it is infeasible based on the path condition computed by DSE. The Choco [10] constraint solver is used to determine satisfiability of those conditions.

It can happen that DSE may not scale to larger methods because of the overhead of recording a path might be high. The future work will analyze literature on path profiling work like one by Ball, et al. [5] to consider efficient alternatives to path recording.

Chapter 6

Evaluation

The goal of the evaluation work is to explore the benefits of the unification framework through a series of experiments. The following section explains a general set up of the experiments by describing the type of analyses and program artifacts used in the experiments. In addition the next section presents an evaluation infrastructure for systematically deriving the constraints θ for conditional static analysis. Afterward in the remaining three sections attempt to answer each of the research questions posed in the beginning of the dissertation.

6.1 Experiment Set Up

6.1.1 Analyses

To answer the three research questions in Sections 6.2-6.4 the experiments are run on the three analyses, SA$_{zero}$, SA$_{sign}$ and DSE, described in the previous chapter. Having two static analyses with comparable domains, i.e., $D_{zero} \prec D_{sign}$, allows us to observe the impact of domain precision on the results.

Since the size of the D_{zero} domain is smaller than the size of the D_{sign} domain, implementing SA$_{sign}$ is significantly harder than SA$_{zero}$. The majority of effort is allocated to implementing transfer functions. For a given expression that takes k variables, where $k \leq 2$ in Soot, a transfer function must be implemented for each possible abstract value of these k variables. Thus, the number of transfer functions grows exponentially with respect to the abstract domain size. Hence, if the results of SA$_{zero}$ are comparable with SA$_{sign}$ then one may determine whether the extra information produced by the more precise analysis is worth the effort of implementing it.

Even though the implemented static analyses are extended to handle `double` data type, the current implementation of DSE lacks this capability. Therefore the results for

asw program are reported for just the static analysis part. If time permits DSE will be also extended.

6.1.2 Programs

The three analyses are run on three subject programs one_tcas [17], wbs [30] and asw [30]. These programs are the Java equivalent of functions used in the NASA aircraft navigation systems. Since the implemented analyses are intra-procedural the functionality of each program was inlined into one method. The main characteristics of the programs are the absence of loops and large numbers of conditional statements in the jimple representation. On the one hand the absence of loops limits the generality of the results, but on the other hand it makes the comparison between PPC of dynamic and static analyses more impartial, because the latter is not disadvantaged by the presence of loops. Table 6.1 provides the number of lines and the number of conditional statements in the jimple representation of each program. Despite the differences in source code size the programs are comparable in terms of branch complexity.

This experiment considers the worst case scenario when an assertion is placed at the end of the method and its variables data-dependent on all other program variable, i.e., A_{slice} would not be able to prune any part of CFG. Thus verifying that assertion is equivalent to analyzing all program paths. If time permits more experiments will be run with different assertions.

6.1.3 Evaluation Infrastructure

In some experiments the evaluation of CSA requires the systematic generation of Π, a set of paths, for the above programs. Since the existing BDD based path generator is not designed for this purpose, an external generator of valid Π is implemented. The external

program	# lines	# conditional stmt
`asw`	347	49
`one_tcas`	160	43
`wbs`	155	45

Table 6.1: Artifacts for the experiments

path generator produces Π with predefined characteristics. For example, a common prefix of length m. A prefix is a set of branch literals that appear consecutively in one of the CFG paths, i.e., Π is a prefix if $\mathcal{L}_\Pi$ does not start with $.^*$, does not have $.^*$ between branch literals, and $\mathcal{L}_\Pi \cap \mathcal{L}_{CFG} = \mathcal{L}_\Pi$.

The generator first builds a *branch dependency* graph that describes the control dependencies between the conditional statements of the CFG. Thus, if the true branch of b_1 leads to b_2 and the false branch of b_1 to b_3 then an edge labeled 1 is added between b_1 and b_2 and an edge labeled 0 is added between b_1 and b_3. Conditional statements b_2 and b_3 have no edges between them since they appear on two different branch outcomes. When branches b_i and b_j appear in sequence, i.e., b_i is not nested in b_j, then an edge labeled 2 is added between them, meaning that on any branch outcome of b_i the conditional statement b_j will execute next.

Next the generator traverses the branch dependency graph by using some directives. For example the building of a prefix of size m starts from the first conditional statement then one of its outcomes is randomly selected. Based on the branch outcome value the graph is traversed to the next conditional statement. This process is repeated m steps. This process is used for generating prefixes for CSA in the experiments described below.

Also, this external generator can be useful in the context of the unification framework. It might be the case that the BDD based Π generator repeatedly produces Π for which an analysis returns an empty PPC, i.e., the analysis is getting ”stuck” with inadequate θ. Then

the external generator might extract a subset of paths from Π by, for example, extending a short prefix.

Even though the external generator produces valid prefixes for CFG, i.e., their paths are in $\mathcal{L}_{CFG}$, it might be the case that they describe infeasible paths. To increase the feasibility of paths produced by the external generator, the BDD store can be used to check if a current prefix has been evaluated before. The current implementation of the external generator lacks this feature.

6.2 RQ1: Comparing CSA and DSE

The experiments in this section attempt to answer the following research question:

> Q1: *How does the effort of generating PPC from partial information from a data-flow analysis compare to the effort required by ideal DSE runs to obtain the same PPC?*

The introductory example for this research question has shown that it may require several DSE runs to obtain the same PPC as a single run of SA. But, in general, situations may also exist where a single run of DSE can obtain the same information as SA. This research question attempts to compare the efforts needed for SA and DSE runs to achieve the same PPC.

Recall that different DSE tools can vary in their implementations, i.e., trace-based or instrumentation-based, and the type of SMT solver used. Therefore beside the execution time, reporting the number of DSE runs and calls to SMT solver are incorporated to better describe the DSE effort.

6.2.1 Experiment Design

The first step in the experiment is to run two SA instances on the three subject programs. Recall that the result of the SA part of the experiment are the set of infeasible branches, i.e., the partial result of SA. Any CFG path containing an infeasible branch trivially conforms to ϕ. However, an infeasible branch by itself cannot describe what part of the CFG should be analyzed by DSE to obtain the same result. Thus, for each program using its branch dependency graph the analyzed part of CFG is determined. This operation is equivalent to the intersection of the language containing the infeasible branch with the language of CFG, call it PPC$_{SA \cap CFG}$.

There are two approaches to using this information to determine whether PPC$_{DSE}$ $=$ PPC$_{SA}$. The first method is to encode PPC$_{SA \cap CFG}$ as a BDD. Then after each DSE run, whose PPC is also encoded as BDD, compare the BDD's of PPC$_{DSE}$ and PPC$_{SA \cap CFG}$ for inclusion. Note, that PPC$_{DSE}$ $=$PPC$_{DSE \cap CFG}$. If after adding the set of satisfying assignment of the PPC$_{SA \cap CFG}$ BDD to the BDD of PPC$_{DSE}$ the set of the satisfying assignment of the latter BDD is the same then DSE has achieved the same PPC as SA. The drawback of this approach is that the result of a DSE run contains branches that are irrelevant to PPC$_{SA \cap CFG}$. For example, if the infeasible branch of PPC$_{SA \cap CFG}$ is b_{10}, t, i.e., true outcome of b_{10} conditional statement, then all branches that appear after it in the trace do not contribute to the target PPC. Such irrelevant branches require extra effort for DSE to process thereby disadvantaging it in a comparison.

An alternative approach is to truncate DSE traces based on PPC$_{SA \cap CFG}$ information. This way only the relevant branches are analyzed and added to BDD. In this case PPC$_{DSE}$ $=$ PPC$_{SA}$ when BDD returns $true$. Checking this adequacy criterion also requires less effort than in the first method. The experiment employs this second method.

program	#infeasible branches		time
	SA$_{zero}$	SA$_{sign}$	
asw	12	19	3.0s
one_tcas	2	2	2.9s
wbs	0	1	3.0s

Table 6.2: Data for the SA part of the experiment

DSE is run twice, first in the search mode to determine a path adequate test suite from a set of test cases, and second in re-run mode. On the second run DSE executes the adequate test suite and collects the report data. Only this second run is counted. Note that this underestimates the cost of a real DSE-based analysis, such as Cute [40], since the test suite generation cost is not counted.

The test cases from which DSE determines the adequate test suite is generated partly from the information of possible domain partitions of the input variables. These facts are obtained by code inspection and manual analyses of DSE traces. Since the deeper an infeasible branch is in the CFG the more domain partitions for input variables are required, i.e., more CFG paths to inspect and test cases to analyze, the timeout for search has been set to 1 hour. The time out cases will be indicated by *.

6.2.2 Results and Analyses

Table 6.2 shows the data for the SA part of the experiment. The first column indicates the program for which the analysis has been performed. The next two column display the number of infeasible branches detected by each type of static analyses. The last column shows the time in seconds that it took for the analysis to produce the result. This time includes PPC generation and BDD construction. In two out of three programs SA$_{sign}$ was able to detect additional infeasible branches.

program	branch	#test cases	search time	test suite	time	SAT calls
one_tcas	22,t	3072	25m	432	11m	1658
	*34,t	6144	60m	488	28m	2718
wbs	27,t	192	70s	144	44s	1726

Table 6.3: Data for the DSE part of the experiment

Table 6.3 describes the results of DSE runs for two programs. The first column displays the program name. The second column identifies the infeasible branch produced by SA. The asterisk indicates that the full adequacy has not been reached for that branch. The third column shows the number of test cases from which DSE searches for a path adequate test suite. The fourth column displays the time it took for DSE to find the adequate test suite. The column "test suite" contains the size of the test suite found within the search time. The penultimate column shows the time it took for DSE in the re-run mode to execute that test suite. The last column shows the number of unique calls to SAT solver. If the prefix of a DSE traces has been analyzed before it was not checked for unfeasibility again.

The comparison of those two tables makes it apparent that the effort required by DSE to obtain the same PPC as SA is greater than the effort required by SA. It takes SA several seconds to obtain the set of infeasible branches, while for DSE in the best case it takes 44 seconds and in the worst case 28 minutes.

For `one_tcas` program DSE has spent the majority of execution time on SAT calls. Even though more SAT were made on `wbs` program the SAT queries generated by `one_tcas` have higher complexity which took by the solver longer to process.

The main challenge in this experiment was to generate a set of test cases of manageable size that contain an adequate test suite. The complexity of finding such a set depends on the number of input variables and the possible partitioning of their input domains. Obviously the deeper the infeasible branch is positioned in the CFG the finer the partitioning of variable input domains, which leads to a larger number of test cases to search in. Thus the

position of the infeasible branch in the CFG also affects DSE effort to achieve the adequate PPC.

Since the cost of DSE run does not include the search time, its results are optimistic. Using concolic execution may reduce the search time of the optimal test suite. But, it will still require the same amount of effort from DSE to analyze infeasible paths.

6.3 RQ2: Comparing the Result of CSA on $\Pi' \subset \Pi$

The second research questions states:

> Q2: *How much additional PPC does a conditional static analysis produce on a set of paths* Π' *compared to the PPC obtained by the same analysis on a larger set of paths* $\Pi \supset \Pi'$*?*

The development of CSA was based on the premise that by analyzing a smaller set of paths CSA performs less data-flow fact merging which is a key source of precision loss in SA. Thus the reduction in merge operations should make the result of CSA more precise. This research question aims to characterize the number of unique infeasible branches generated from execution prefixes of different lengths, since the longer the prefix the fewer merges CSA performs. Thus, the measure of PPC is unique infeasible branches, as before, and Π is manipulated by varying prefix length.

6.3.1 Experiment Design

The PPC of the experiment is evaluated as the number of *new* infeasible branches detected by CSA. An infeasible branch b_i of a prefix of size m is considered to be new if b_i cannot be determined by any proper subset of prefixes of that prefix, i.e., of sizes $m-1$. The relation between the path prefixes of size $m-1$ and m is described by $\Pi' \subset \Pi$ where Π'

is described by the shorter prefix and Π by the longer prefix. Comparing the number of infeasible branches between different prefix sizes will answer the research question.

A complete characterization of the path subset relation requires the exhaustive traversal of P's branch dependency graph to generate all possible paths. Since the number of paths can become exponentially large such a comprehensive method for path generation is not suitable. Instead the branch dependency graph is used to randomly generate distinct paths. Next, a CSA is run with every prefix of each generated path. To make sure that the same prefix is not analyzed twice the implementation tracks the set of analyzed prefixes. If a prefix of the length k has not been analyzed yet then its result, i.e., the number of infeasible branches, is added to the result for the prefixes of the length k.

For example, consider the path l_1, l_2, l_3. First the analysis is run on the path's prefix l_1 and its result is recorded for $k = 1$. Next, the analysis is run on l_1, l_2 and for that prefix only those infeasible branches are recorded that the previous runs, i.e., with l_1, was not able to produce. Finally CSA is run with the l_1, l_2, l_3 prefix and the new infeasible branches are added to the prefixes of that size. If the next path to be analyzed is described by $l_1, \neg l_2$ then l_1 will not be run by CSA since that prefix has been evaluated before.

In the experiment 1000 distinct paths have been generated for each program. The maximum length of generated paths was 35 for `asw`, 36 for `one_tcas` and 45 for `wbs`. To ensure that the experiments can be reproduced the paths for each program first were generated and written to a file. Then these paths are read from the file and each path is evaluated in the manner described in the previous paragraph.

6.3.2 Results and Analyses

Tables 6.4-6.6 show the results of the experiments for the three programs. The first column of each table describes the length of the prefix analyzed by CSA. The second column

"#all" shows the total number of unique prefixes of such length, e.g., there can be only two prefixes of length one which are l_1 and $\neg l_1$. The third column "#prefixes" displays the number of prefixes of each length for which CSA was able to produce infeasible branches. If the result of CSA$_{sign}$ is different from CSA$_{zero}$ its result is shown in parentheses. The next column "#infeasible" contains the total number of infeasible branches produced by the prefixes in the previous column.

In the experiment setting Π is represented by a row i and Π', which is a proper subset of Π, is represented by any $i + k$ row where $k \geq 1$. Thus, row 4 contains the result for prefixes of length 4 and row 5 for prefixes of length 5 which subsume the prefixes of row 4. Moving from one row down to the next corresponds to subsetting the paths.

For example, consider the data for the prefix of length 13 in Table 6.4. The line indicates that the total unique prefixes of that length is 945, but for only 36 of them is CSA$_{zero}$ able to calculate the 292 infeasible branches. Note, that 292 branches are not necessary unique since the prefixes of the same length might find the same infeasible branches. For the same set of prefixes CSA$_{sign}$ was able to find infeasible branches for only two prefixes, in parentheses in row 13, and there were on average seven per each such prefix.

This result might seem contradictory since CSA$_{sign}$ must detect at least all infeasible branches that CSA$_{zero}$ does. This is because the relation between these two infeasible branch sets is not apparent from the data. If for a given prefix CSA$_{sign}$ finds an infeasible branch leading to some part of the CFG, that part of CFG is not analyzed by CSA$_{sign}$. But if CSA$_{zero}$ fails to mark this branch infeasible then it will analyze that part of CFG and possibly detect infeasible branches there.

Each table only shows the data up to the last prefix which produces some result for at least one of the analysis. Therefore, CSA fails to find additional PPC for `asw` and the prefixes with length 19 to 35. For `one_tcas` this range is between 20 and 36, and for `wbs` this range is between 22 and 45. Thus, CSA does not produce results for paths

corresponding to longer prefixes and as the data indicate the length should be limited to about half of the maximum path length.

The data clearly show that restricting the path to be analyzed, i.e., $\Pi \subset \Pi'$., can yield additional precision. However, the increase is not consistent among programs. Constraining paths for `wbs` produces on average between 2 and 4 new infeasible branches. While for `one_tcas` this number is between 1 and 3. The identification of new infeasible branches for `asw` is not that consistent. At some prefixes an analysis can determine up to 8 infeasible branches, but the majority of larger prefixes produce no new information.

Even though longer prefixes can allow for CSA to find additional PPC there are only a small fraction of those prefixes do so compared to the available prefixes, e.g., 36 out of 945. This suggests that additional information generated by the analysis or by an external data dependency analyzer can be used to detect prefixes of the same length that might produce additional PPC, i.e, generating a good θ. For example CSA can track the merge points on which it looses precision. Then it makes this information available to a prefix generator which choses prefixes that eliminate those merge points.

In this experiment only paths that can be described by a prefix have been considered. However, the path language is much richer and can describe such path subsumption relation as $.^{*}; b_n; .^{*}$ and $.^{*}; b_n; .^{*}; b_k$. In order to fully comprehend the $\Pi \subset \Pi'$ relation a more diverse set of subsumption relations should be analyzed.

6.4 RQ3: Comparing a Single Analysis and a Set of Analyses

The third research question of this dissertation is:

length	#all	#prefixes	#infeasible
0	1	1	12(19)
1	2	2	16(21)
2	4	0	0
3	8	0	0
4	16	0	0
5	32	10(4)	20(4)
6	64	0	0
7	128	0	0
8	250	0	0
9	424	45(8)	93(8)
10	628	0	0
11	783	0	0
12	894	0	0
13	945	36(2)	292(14)
14	967	0	0
15	982	0	0
17	995	0	0
16	991	12(1)	12(1)
19	998	1(0)	5(0)
18	998	2(0)	10(0)

Table 6.4: RQ2 results for `asw`

length	#all	#prefixes	#infeasible
0	1	2	2
1	2	2	3
2	4	4	5
3	8	0	0
4	16	2	2
5	32	4	4
6	64	4	8
7	128	0	0
8	253	2	2
9	463	2	8
10	656	9	38
11	724	13	47
12	666	10	41
13	559	5	18
14	444	0(2)	0(2)
15	387	2	4
16	361	0	0
17	351	0(1)	0(1)
18	350	1	1
19	350	1	2

Table 6.5: RQ2 results for `one_tcas`

length	#all	#prefixes	#infeasible
0	1	0(1)	0(1)
1	2	2	12(14)
2	4	2	2
3	8	3	3(4)
4	16	3	8
5	32	6	26(27)
6	64	10	40(44)
7	128	9	41(46)
8	254	15	58(62)
9	447	20	72(77)
10	639	9	26(28)
11	800	7	14
12	900	4	8
13	948	2	4
14	969	1	1(2)
15	982	0	0
17	992	2	2(4)
16	991	2	2(4)
19	945	0	0
18	978	0	0
21	833	1(0)	1(0)

Table 6.6: RQ2 result for `wbs`

Q3: *Given a fixed effort, how does the PPC of a single analysis compare to the combined PPC of a set of diverse analyses?*

The first research question has shown that DSE can produce the same result as CSA even though achieving the same PPC requires additional DSE runs. However, as the second research question indicates, CSA was able to produce new information only for a few prefixes. Thus, on the one hand several DSE runs can produce the same result as a single run of CSA, but on the other hand CSA cannot produce a new partial result for every θ. Since the recourse allocation for DSE or CSA runs is depended on the analyses implementation, the evaluation assumes equivalent effort required for a single DSE or CSA run. Given a budget of N runs, i.e, the effort measure, one needs to decide how to allocate them between DSE and CSA to optimize PPC. If all N runs are given to DSE then the analysis might immerse itself in one part of CFG by meticulously exploring it. If all N runs are to be allocated to CSA then it might analyze many prefixes without increasing PPC. This research question explores the changes in PPC when a program is analyzed by the combination of DSE and CSA where $N/2$ runs are allocated to DSE and another $N/2$ to CSA. The $N/2$ was chosen as starting point for further experiments. Then the combined result is compared to PPC of DSE and CSA running N times in isolation.

6.4.1 Experiment Design

First each analysis is run separately for `one_tcas` and `wbs` programs. The effort N was chosen to be 1000 for `one_tcas` and 400 for `wbs` is 400. This choice was dictated by the complexity of the input domain of each program.

DSE is executed on a random set of inputs, which are unique but do not necessarily produce unique paths. The result of DSE runs are recorded into the BDD store PPC$_{dse}$.

The results of DSE run were recored after $N/2$,i.e, PPC$_{dse/2}$, runs for use in the combining analysis part of the experiment.

Each of CSA$_{zero}$ and CSA$_{sign}$ were run in two modes: random and controlled. In the random mode, CSA$_{random}$, θ is a prefix of random length m. Such a prefix is generated by randomly traversing the branch dependency graph up to depth m. The total of 1000 prefixes of various length were generated for `one_tcas` and 400 for `wbs`. The result was recorded into the BDD store, PPC$_{sar}$ of each analysis. Also after the half of the prefixes have been analyzed CSA$_{random}$ records the PPC value, i.e., PPC$_{sar/2}$ into a separate BDD store to be combined with PPC$_{dse/2}$.

In the controlled mode CSA$_{controlled}$ obtains θ directly from the BDD store. The unexplored prefix is obtained from the BDD store by querying BDD for its unsatisfying cubes, i.e.,"paths" leading to the *zero* node. If CSA$_{controlled}$ cannot produce PPC for all prefixes acquired from the BDD store, then it uses one of the prefixes generated for CSA$_{random}$. It executes those prefixes in the same order as CSA$_{random}$. The total number of CSA$_{controlled}$ is executed 1000 for `one_tcas` and 400 for `wbs`.

The combining of $N/2$ of DSE and $N/2$ CSA analyses are done differently for CSA$_{random}$ and CSA$_{controlled}$. The combining of PPC$_{dse/2}$ and the random CSA$_{csa/2}$ is not interactive. The corresponding BDD stores of two analyses that were recorded half way through are combined resulting in PPC$_{dse/2+csa/2}$.

CSA$_{controlled}$ combines its data in an interactive way. First, the BDD store is initialized by PPC$_{dse/2}$. Next CSA$_{controlled}$ queries that BDD store for the unexplored prefixes. However, this time it runs only half of the initial runs, i.e., 500 for `one_tcas` and 200 for `wbs`. The same PPC$_{dse/2}$ is used for both random and controlled analyses.

As discussed previously, shorter satisfiable paths in the BDD store correspond to larger subtree coverage in the program's execution tree. Comparing the distribution of prefix sizes is suitable when PPC of two analyses are combined into one and that result is compared to

PPC of one of the two initial analyses. This is because the subsumption relation between the cumulative and separate results is apparent. However, comparing PPC of two analyses with an unknown subsumption relation among paths cannot be done based on the prefix size distribution. A prefix of a smaller size of one analysis may contain prefixes of larger size of another analysis. For example l_1, l_7 prefix contains l_1, l_2, l_7 and l_1, l_3, l_7 prefixes, but without such knowledge one prefix of size two and two prefixes of size 3 are incomparable.

In the case when a subsumption relation is unknown, the comparison between PPC_1 and PPC_2 is done based on the number of unique prefixes of each PPC. A unique prefix of PPC_1 is not contained in any prefixes of PPC_2 and vice versa. A prefix of PPC_1 is unique if by adding that prefix to the BDD store of PPC_2 the number of prefixes, i.e., satisfying assignments, in the BDD store of PPC_2 changes. If the number of prefixes in the BDD store of PPC_2 does not change then the prefixes have been subsumed by one of the prefixes in that BDD store. PPC with the larger number of unique shorter prefixes corresponds to higher subtree coverage than PPC with the smaller number of unique prefixes, or longer prefixes

Thus the comparison of any of two combined analysis with DSE or CSA is done by identifying the number of unique prefixes for each pair, i.e., combined versus DSE and combined versus CSA.

6.4.2 Results and Analyses

The results are presented on Figures 6.1 and 6.4 for SA_{zero} and SA_{sign} domains respectively. The y-axis is the length of the prefix and x-axis represent the number of prefixes of particular length. Each graph shows the distribution of unique prefixes. The solid line “DSE vs. Combined” describes the unique prefixes of DSE analysis compared to the combined analysis. The dotted line “CSA vs. combined” shows the unique prefixes of CSA

analysis compared to the combined analysis. The dashed line "Combined vs. DSE" represents the number of unique prefixes of the combined analysis compared to the prefixes of DSE. The dot-dashed line "Combined vs. CSA" describes the number of unique prefixes of combined analysis compared to the CSA prefixes.

Each graph is identified by the type of program, i.e., TC for `one_tcas` and WBS for `wbs`, the type of analysis CSA_{zero} or CSA_{sign}, and by how θ was generated for the analysis, i.e., random or controlled. Thus Figure 6.1[a] shows the result for `one_tcas` program executed by CSA_{zero} analysis in random mode. To compare the combined analysis, which is in this case half CSA_{zero} with randomly generated θ and half DSE runs, consider the solid and the dashed lines. The comparison between solid and dashed of Figure 6.1[a] indicates that there were fewer unique prefixes in PPC_{dse} comparing to $\text{PPC}_{dse/2+csa/2}$ and there were many unique prefixes in $\text{PPC}_{dse/2+csa/2}$ of the combined analysis compared to the PPC_{dse} of DSE. This trend appears on all graphs and especially emphasized when CSA is run in the controlled mode. This implies that effort spent on running combined analysis produces higher PPC than effort spent on running single DSE analysis.

To compare the combined analysis with CSA consider the dotted line and the dot-dashed line. One the same Figure 6.1[a] the dotted line indicates that there were fewer unique prefixes in CSA_{zero} comparing to the prefixes of the combined analysis. The dot-dashed line depicts larger number of unique prefixes of shorter length. However, the difference is not as dramatic as in the case of DSE and the combined analysis. The rest of the graphs exhibit the same trend that running the combined analysis result in more unique short prefixes than the prefixes of CSA. Again, this difference is more apparent when CSA is run in the controlled mode.

The inclination of the combined analysis to have larger number of unique prefixes than DSE or CSA does not depend on the strength of CSA, which points to the benefits of the

technique. Thus, combining the result of any other analyses should also result in a better PPC coverage as compared to a single analysis running for the same amount of time.

The combining of analysis is beneficial even when the analyses are run independently of each other, i.e., in the absence of control. As the data indicate, running analyses where CSA is controlled provides even larger differences in PPC. One can expect even better PPC coverage when the control for DSE is implemented. Then different interaction schedules can be explored, e.g., one DSE run is followed by one CSA run.

6.5 Summary and Limitations of Results

The first experiment has shown that DSE requires more effort than CSA to obtain the same PPC as CSA. This result can be generalized only to PPC produced by CSA. However, this result cannot be extended to an arbitrary PPC because CSA may not be powerful enough to obtain that PPC, while given enough time DSE might achieve such coverage. For programs with loops DSE might never achieve the given PPC if it describes paths that contain loops.

The second experiment has indicated that CSA can become more precise when it considers smaller numbers of paths. However, the increase in the precision does not manifest for any subset of paths. This result can only be generalized to paths with common prefixes, however additional study should be run to determine the results for other $\Pi \subset \Pi'$ relations. Moreover, it is unknown if the results are applicable to programs with loops.

The third experiment has shown that allocating the effort between two different analyses increases the number of unique BDD prefixes compared to the number of unique prefixes produced by a single analysis utilizing the same amount of effort. The main threat to validity of this experiment is that the characterization of the result in just terms of unique prefixes might not be descriptive enough. For example, two prefixes of the same length might correspond to different number of paths in CFG. Thus the unique prefixes of the

same length might carry different weight for a tester. Future studies should consider other attributes of the unique prefixes beside its length that can better quantify PPC.

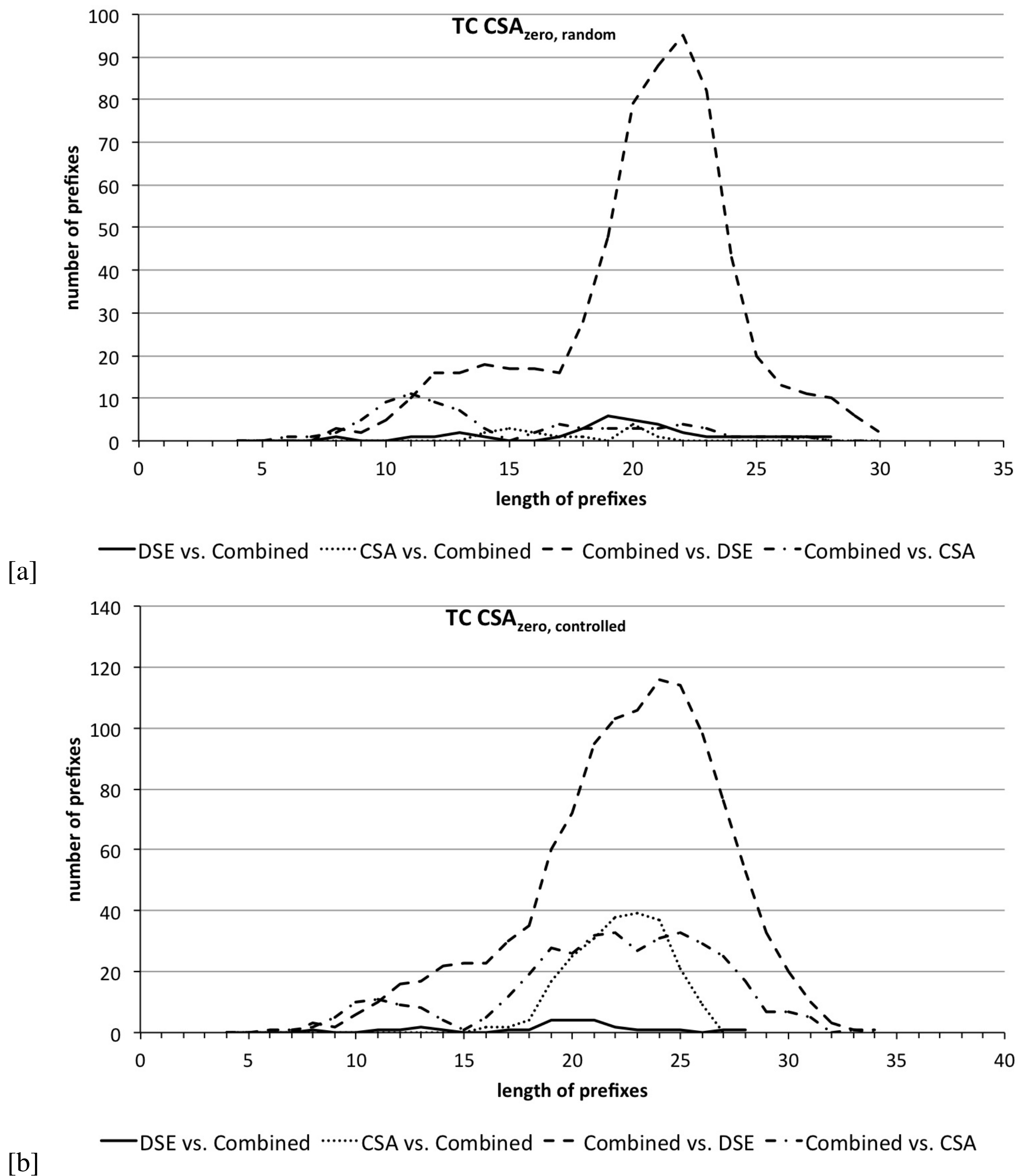


Figure 6.1: RQ3 results for CSA$_{zero}$ and `one_tcas`

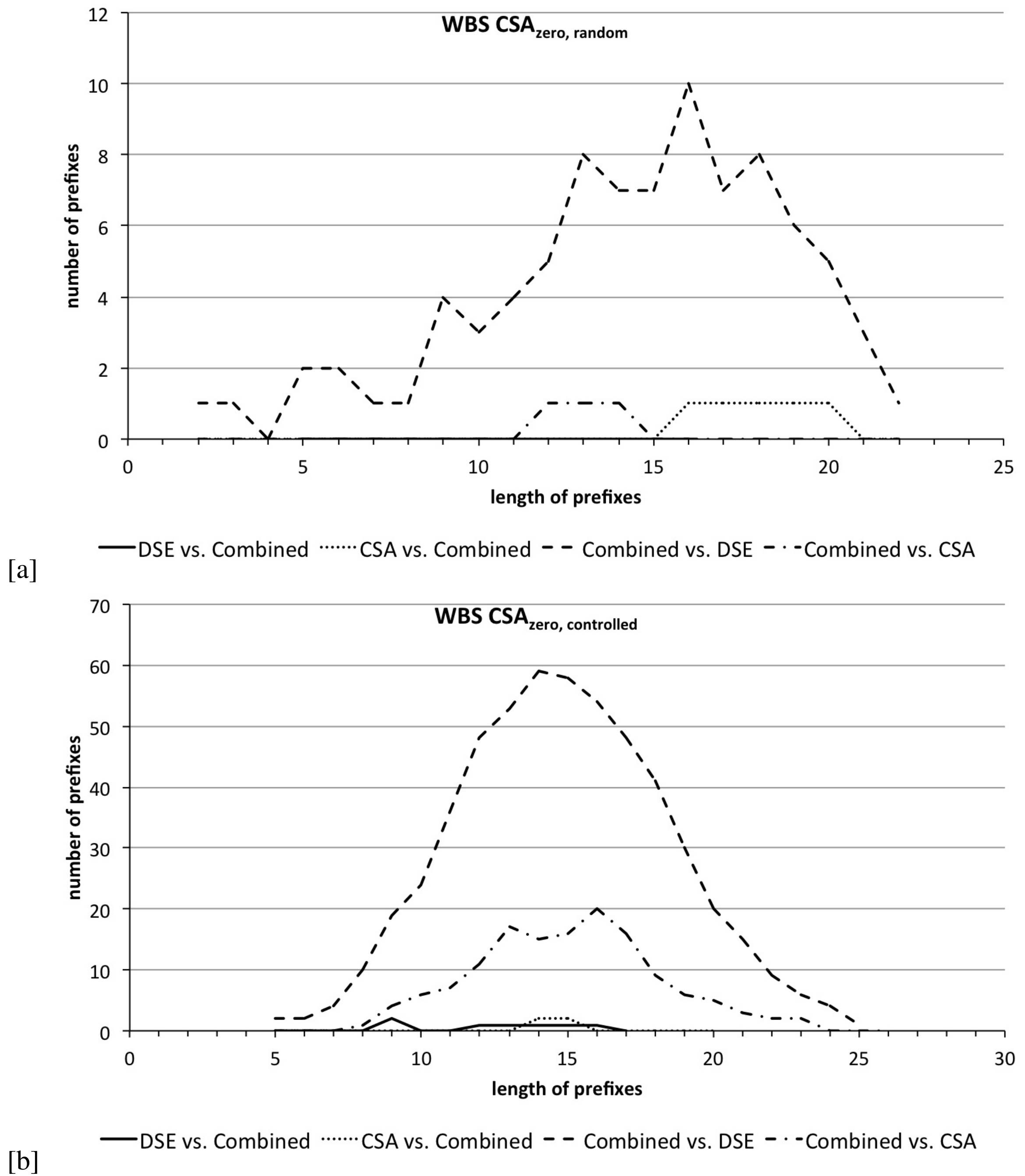


Figure 6.2: RQ3 results for CSA_{zero} and `wbs`

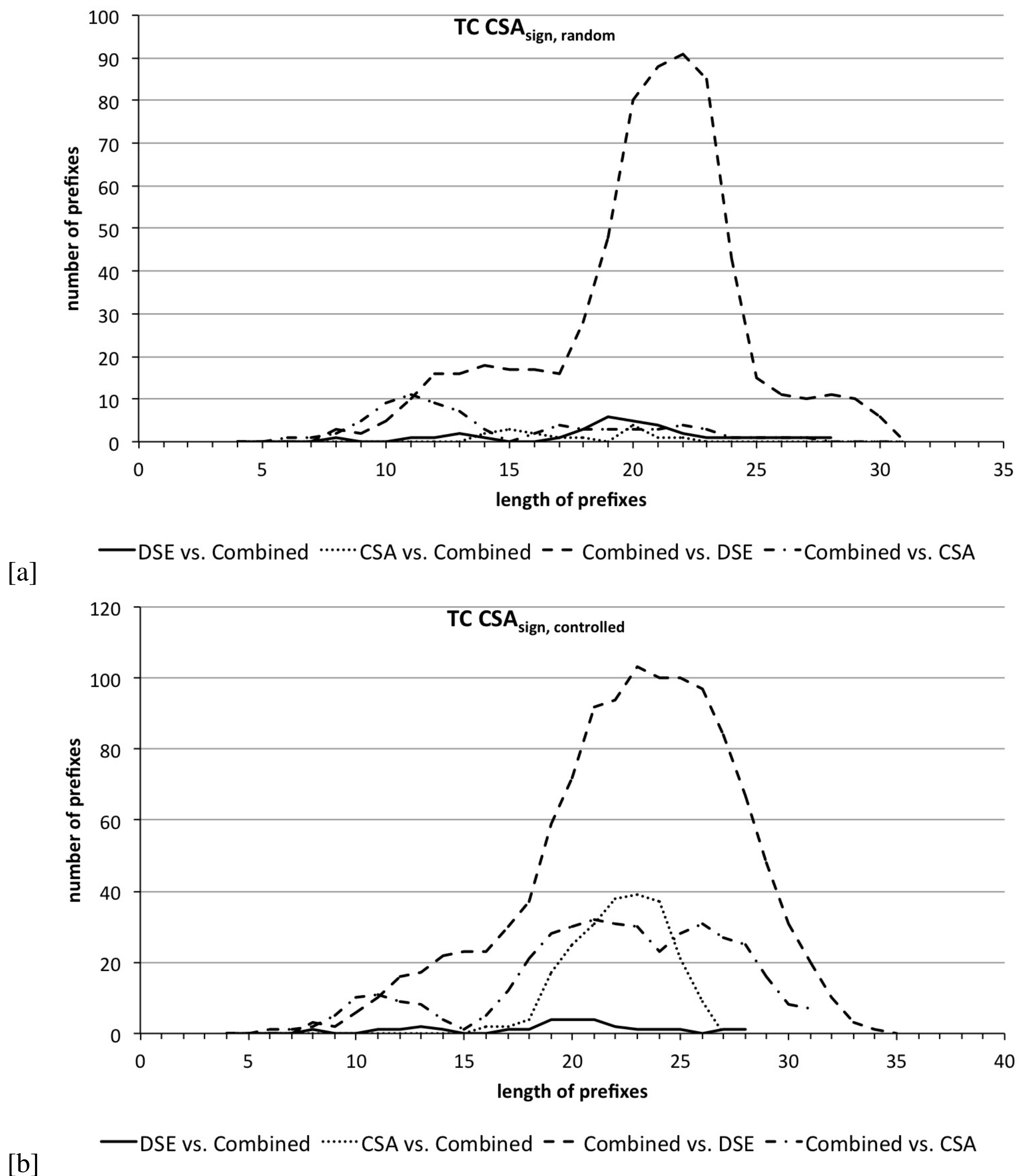
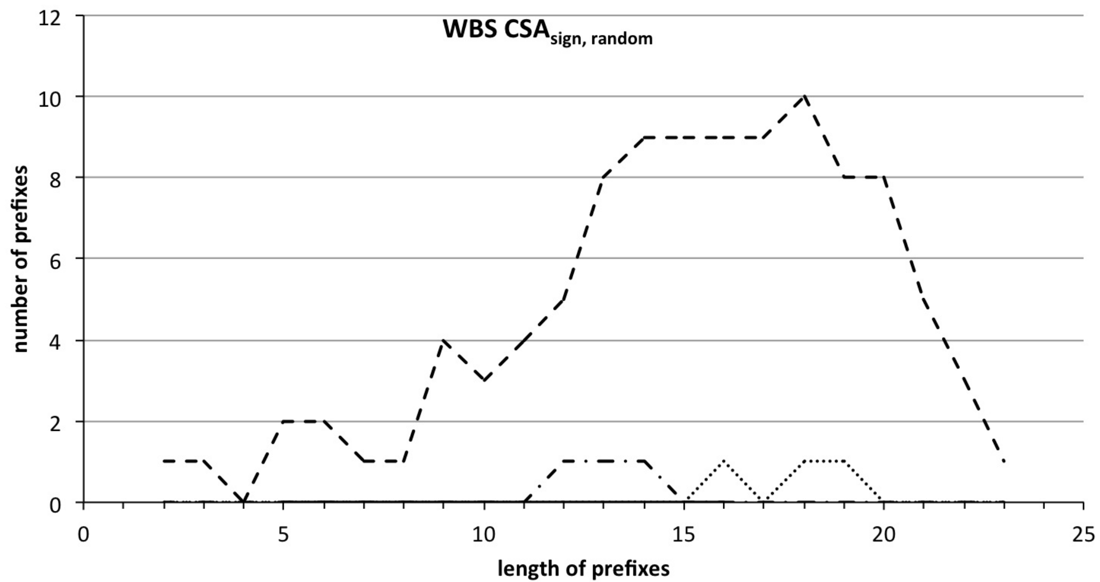
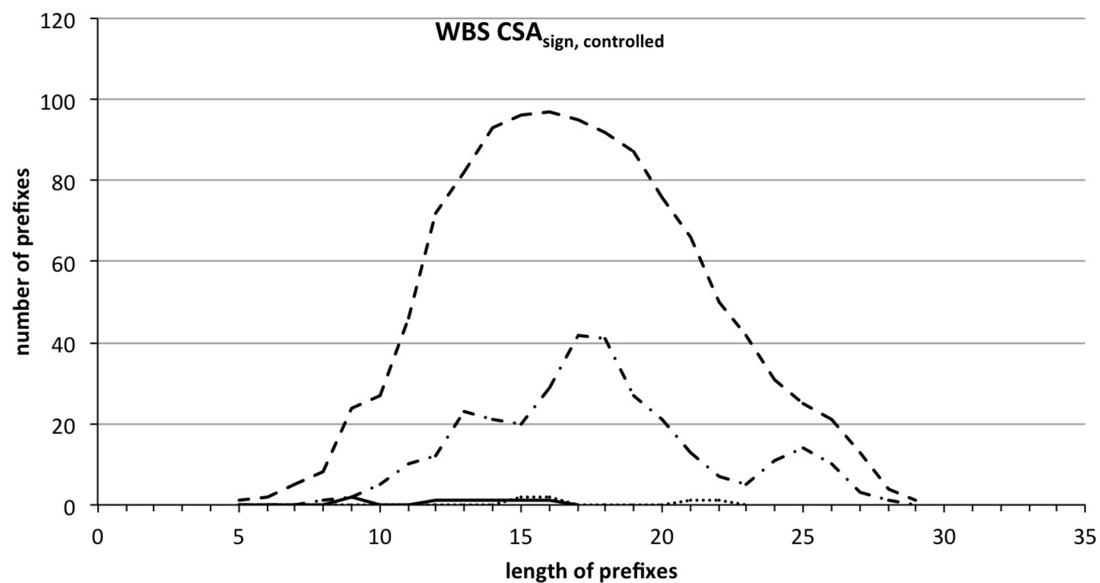


Figure 6.3: RQ3 results for CSA_{sign} and `one_tcas`



[a]



[b]

Figure 6.4: RQ3 results for CSA$_{sign}$ and `wbs`

Chapter 7

Conclusion

The benefits of combining different analyses have been known for a long time and are widely used in the state of the art program analyses tools. From the theoretical work of Patrick and Radhia Cousot [14] on combining static analyses to the resent implementations of the SPIN model checker [39], DSD-crasher [15], the Yogi project [6, 25, 34], where static and dynamic analyses work in cooperation to decide whether $P \models \phi$, research has shown the positive results of different analyses working together. In those implementations there is usually one main analysis, e.g., a model checker, and several auxiliary analyses, e.g., a liveness analysis, that help the main analysis to produce its final result.

This dissertation considers the undertaking of combining analysis in a different context. Instead of combining intermediate results of analyses, their final results, which analyses may have computed under some condition, are combined. Hence, the dissertation work is more in line with work on conditional soundness [13] and conditional analysis [8] where the results of analyses are correct only under some conditions. The dissertation extends the notion of conditional analysis by applying it to the data-flow analysis framework where conditions are described by a set of paths on which an analysis can decide $P \models \phi$. The stronger the condition the fewer program paths it represents. Thus, in the context of conditional analyses, combining analyses reduces to combining their conditions.

Conceptually, a single analysis, e.g., a dynamic analysis, can analyze all program paths by considering a single path at a time. Conditional static analysis can do the same. The main question that the dissertation explores is whether several analyses can analyze paths more efficiently than each analyses running in isolation.

To answer this question the dissertation conducts a series of evaluations. The first experiment shows that it takes much longer for a dynamic analysis to analyze the same set of paths as compared to a conditional static analysis. The second experiment shows that unlike dynamic analysis, conditional static analysis cannot produce even partial result for each set of paths. These two evaluations highlights the disadvantages of one analysis

in comparison to another. Ultimately the third evaluation shows that running two different analyses results in a greater number of analyzed paths than each analysis can achieve for the same amount of effort.

In order to answer the research questions that support the thesis statement the dissertation has developed the unification framework which incorporates different analyses. Also, the dissertation explains how dynamic and conditional static analyses can be extended for integration into the unification framework. In addition, the dissertation introduced the language for describing program paths. Using this path language, program paths can be identified without enumerating them. In order to efficiently manipulate program paths, e.g., to combine them, the dissertation proposes a BDD based encoding for a program's path language. Such an encoding can be queried for a set of paths that have and have not been analyzed. This information is used by the unification framework for its reporting and controlling capabilities.

7.1 Future Work

The future work will augment the unification framework with additional analyses and test its scalability on a richer set of programs. The generality of the unification framework can be examined by adding controlling and reporting capabilities to such analyses as model checking and symbolic execution.

One direction of the future work is to further develop the subtree adequacy criterion. In testing, the path adequacy criterion is less used than any other structural adequacy criteria. One of the main drawbacks of using path coverage is the unknown number of feasible paths in the program. This prevents a meaningful reporting of the degree to which a test suite is path adequate. Moreover, visualizing a set of paths that have been covered or uncovered is more challenging than other structural criteria as statements. Thus, developing further the

notion of subtree coverage and projection of covered subtrees on the CFG would provide a tester with intuitive information on program path coverage. A tester might decide to have some part of the CFG be path adequately tested while other parts of CFG to be tested using a weaker adequacy criteria. The subtree coverage criteria and its visualization can provide mechanisms for achieving that task.

Another direction of the future work is extending the framework to handle multiple properties. A program might have several properties to which it should conform. However, the current unification framework is designed to handle a single property. Using the unification framework to verify each property separately can lead to repetitive work when analyses consider the same set of paths several times. Thus, extending the unification framework to handle multiple properties may be beneficial. In addition to verifying simultaneously several properties the unification framework can use additional information like the relation between properties for more efficient verification of new properties.

A final direction of future work is combining conditional static analyses in its traditional sense, i.e., by the means of intermediate results. The combining of two analyses will produce more precise results only if the intermediate results of each analyses are not informative. Since the values of the intermediate results computed by a conditional static analysis are more precise than the values of calculated by traditional static analyses then combining conditional static analyses may result in more powerful analyses. For example, if a conditional analysis cannot verify ϕ for some set of paths then it produces some partial result, but if the intermediate results of that conditional analysis are to be combined with the intermediate result of another conditional analysis over the same set of paths, then they might decide the property ϕ for that set of paths.

Bibliography

[1] Frances E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[2] Saswat Anand, Corina Păsăreanu, and Willem Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer (STTT)*, 11:53–67, 2009. 10.1007/s10009-008-0090-1.

[3] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.

[4] Shay Artzi, Adam Kieżun, Jaime Quinonez, and Michael D. Ernst. Parameter reference immutability: formal definition, inference tool, and comparison. *Automated Software Engg.*, 16(1):145–192, March 2009.

[5] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[6] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 3–14, New York, NY, USA, 2008. ACM.

[7] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[8] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking. *CoRR*, abs/1109.6926, 2011.

[9] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35:677–691, August 1986.

[10] Choco. `http://choco-solver.net`.

[11] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50:752–794, September 2003.

[12] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17:181–196, March 1995.

[13] Christopher L. Conway, Dennis Dams, Kedar S. Namjoshi, and Clark Barrett. Pointer analysis, conditional soundness, and proving the absence of errors. In *Proceedings of the 15th international symposium on Static Analysis*, SAS '08, pages 62–77, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proc. Symp. on Princ. of Prog. Lang.*, pages 269–282, 1979.

[15] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17:8:1–8:37, May 2008.

[16] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international con-*

ference on Software engineering, ICSE ’08, pages 281–290, New York, NY, USA, 2008. ACM.

[17] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10:405–435, October 2005.

[18] Mark Doliner. Cobertura. http://cobertura.sourceforge.net, 2005.

[19] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. Joining dataflow with predicates. *SIGSOFT Softw. Eng. Notes*, 30(5):227–236, September 2005.

[20] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. *SIGPLAN Not.*, 37(1):191–202, January 2002.

[21] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10):1483–1498, October 1988.

[22] gnu.org. gcov. http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Gcov.html, 2012.

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[24] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV ’97, pages 72–83, London, UK, UK, 1997. Springer-Verlag.

[25] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. Synergy: a new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT ’06/FSE-14, pages 117–127, New York, NY, USA, 2006. ACM.

[26] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. *SIGPLAN Not.*, 41(6):376–386, June 2006.

[27] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13:315–353, 2000.

[28] W.C. Hetzel and B. Hetzel. *The Complete Guide to Software Testing*. John Wiley & Sons, 1988.

[29] JDD. `http://javaddlib.sourceforge.net/jdd`.

[30] JPF. `http://babelfish.arc.nasa.gov/trac/jpf`.

[31] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[32] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 327–338, New York, NY, USA, 2007. ACM.

[33] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[34] Aditya V. Nori and Sriram K. Rajamani. An empirical study of optimizations in yogi. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 355–364, New York, NY, USA, 2010. ACM.

[35] N. Parrington and M. Roper. *Understanding software testing*. Computers and their applications. E. Horwood, 1989.

[36] Corina S. Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proc. of Intl. Conf. on ASE*, pages 179–180, 2010.

[37] Vlad Roubtsov. Emma. http://emma.sourceforge.net/index.html, 2006.

[38] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proc. of ICSE*, pages 341–350, 2008.

[39] Joel P. Self and Eric G. Mercer. On-the-fly dynamic dead variable analysis. In *Proceedings of the 14th international SPIN conference on Model checking software*, pages 113–130, Berlin, Heidelberg, 2007. Springer-Verlag.

[40] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[41] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[42] Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .net. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[43] Ashish Tiwari and Sumit Gulwani. Logical interpretation: Static program analysis using theorem proving. In *Proceedings of the 21st international conference on Automated Deduction: Automated Deduction*, CADE-21, pages 147–166, Berlin, Heidelberg, 2007. Springer-Verlag.

[44] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proc. of CASCON*, pages 13–. IBM Press, 1999.

[45] E. J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Commun. ACM*, 31(6):668–675, June 1988.