An Efficient 2-Phase Strategy to Achieve High Branch Coverage

Sarvesh Prabhu

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

Master of Science
in
Computer Engineering

Michael S. Hsiao, Chair
Sandeep K. Shukla
Yaling Yang

3 February, 2012
Blacksburg, Virginia

Keywords: Branch Coverage, Conflict-driven Learning,
Symbolic Execution, Software Testing

An Efficient 2-Phase Strategy to Achieve High Branch Coverage

Sarvesh Prabhu

(ABSTRACT)

Symbolic execution-based test generation is gaining popularity for software test generation. The increasing complexity of the software program is posing new challenges in software execution-based test generation because of the path explosion problem. We present a new 2-phase symbolic execution driven strategy that achieves high branch coverage in software quickly. Phase 1 follows a greedy approach that quickly covers as many branches as possible by exploring each branch through its corresponding shortest path prefix. Phase 2 covers the remaining branches that are left uncovered if the shortest path to the branch was infeasible. In Phase 1, a basic conflict driven learning is used to skip all the paths that may have any of the earlier encountered conflicting conditions, while in Phase 2, a more intelligent conflict driven learning is used to skip regions that do not have a feasible path to any unexplored branch. This results in considerable reduction in unnecessary SMT solver calls. Experimental results show that significant speedup can be achieved, effectively reducing the time to detect a bug and providing higher branch coverage for a fixed time out period than previous techniques.

$\sim$ *To my Grandparents* $\sim$

Acknowledgments

I would, first of all, like to thank my research advisor, Dr. Michael S. Hsiao, for his selfless guidance and support during my research. His exceptional teaching in the course, Testing of Digital Systems, inspired me to join his PROACTIVE research group. I was honoured to get a chance to work with him and was deeply inspired by his extensive knowledge, dedication and amiable nature. I would like to thank Dr. Sandeep Shukla and Dr. Yaling Yang for serving on my thesis committee.

My sincere thanks also goes to Loganathan Lingappan, Vijay Gangaram and Jim Grundy from Intel Corporation for funding this research. Their suggestions during our meetings immensely helped me in this research work.

I also thank my lab-mate Saparya Krishnamoorthy for taking time out of her own crunch time to explain the concepts of symbolic execution and answering all my doubts.

Special thanks to my roommates Supratik Misra, Anup Mandlekar, Apoorv Naik and Akshay Sankpal who have made my stay at Virginia Tech truly memorable. I would also like to thank my friends Sachin Hirve, Amrapali Dengada and Kavya Shagrithaya for all the curricular and non-curricular discussions that we had.

I would like to express my gratitude to my parents Pradeep Prabhu and Vidhya Prabhu, my sister Sheetal Prabhu and my extended family for their love, support and encouragement at every step of my life.

I thank my PROACTIVE lab-mates Dhrumeel Bakshi, Neha Goel, Nikhil Rahagude, Maheshwar Chandrasekhar, Mainak Banga, Chinmay Limaye, Min Li, Huy Nguyen, Indira Priyadarshani and Nevedetha Narayanan for helping me throughout and for all the fun we had over the last two years.

Sarvesh Prabhu

February 2012

Contents

1 Introduction **1**

1.1 Application to Tester Programs 2

1.2 Contributions 5

1.3 Thesis Organization 6

2 Background **7**

2.1 Symbolic execution-based test generation 8

2.1.1 Program Instrumentation 11

2.2 Concolic execution 16

2.3 Satisfiability Modulo Theory (SMT) solvers 17

2.4 Previous work 19

2.4.1 DFS-based reachability-guided strategy 21

2.4.2 Disadvantages of the DFS based reachability-guided strategy 23

3 Phase 1 **27**

3.1 The Strategy for Phase 1 27

3.1.1 Lower coverage reported 30

3.2 Conflict Driven Learning 32

3.2.1 UNSAT core 32

3.2.2 Advantages of Proposed Phase 1 34

4 Phase 2 with Intelligent Conflict-driven Learning **36**

4.1 Need for an intelligent conflict driven learning 37
4.2 Phase 2 Algorithm with intelligent conflict-driven learning 39

5 Experimental Results **48**

6 Conclusion and Future Direction **53**

Bibliography **55**

List of Figures

2.1 Example of a Control Flow Graph 11

2.2 A reachability graph where the DFS based reachability guided strategy explores a node through a longer path even though shorter paths exist 24

2.3 A reachability graph where a DFS based strategy gets confined in a subspace for a long time 25

3.1 An example of procedure followed in Phase 1 29

3.2 A reachability graph showing the need to report low coverage in Phase 1 . . 31

4.1 A reachability graph showing the need for an intelligent conflict driven learning in Phase 2 37

4.2 A reachability graph showing the use of *no feasible path nodes list* 45

4.3 A reachability graph showing a totally unexplored node 47

List of Tables

1.1 SKUs associated with Intel® Core™ desktop processor family 3

2.1 Example of Symbolic Execution 9

5.1 Phase 1 Results 49

5.2 Phase 2 Results 50

List of Algorithms

2.1 DFS-based reachability-guided search strategy 22

3.1 Algorithm for Phase 1 28

4.1 Algorithm for Phase 2 40

4.2 Algorithm for *reachability check* of branch (n, V) 40

Chapter 1

Introduction

Testing of software plays a critical role in ensuring the quality of the product, and nowadays testing accounts for about 50% of the development cost [1]. According to a study by the National Institute of Standards and Technology [2], software bugs or errors are so prevalent that they cost the US economy an estimated $59.5 billion annually. The study also found that more than a third of these costs could be saved by improving the testing infrastructure. For effective testing of a software program, the desire is that all possible behaviors of the program can be exercised by the test suite. This is not achievable by applying random inputs or manual test generation. Random inputs tend to exercise the same execution paths while hard to reach paths remain untested.

Various techniques [3–5] have been developed for automated test generation for software. The advanced method of symbolic evaluation can be applied to program testing situations with results close to those of formal correctness proofs, but without the high cost of generating such proofs [6]. SELECT[7] and EFFIGY[8] were the first systems that demonstrated symbolic execution. In symbolic execution, the path of the program which needs to be exercised is converted to a logical formula. This formula is then given to a constraint solver which returns an input that exercises the path, if such an input exists. On the other hand, if no input exists that can exercise the path, the solver concludes that the path is infeasible. The symbolic execution based methods are guaranteed to make conclusions about distinct paths of the program under test.

Even though symbolic execution based methods are very effective in path based test generation, these methods currently do not scale to large programs. The number of paths increase exponentially with increase in the size of the program. Hence, it is impossible to exercise every path of the program in a short time. This problem is usually referred to as the *path explosion problem*. Therefore, instead of exploring all paths in a program code, other metrics that can relate the all the paths are used. These metrics include statement coverage, function coverage, branch coverage, etc. For instance, 100% branch coverage and 100% statement coverage indicates that all conditions and statements in the program have been exercised. This can give a notion of how much the program has been tested.

1.1 Application to Tester Programs

In order to achieve high yields and satisfy the ever-changing market demands, the number of SKUs (Stock Keeping Units) associated with a micro-processor product has risen sharply over the last decade. This is also driven by the number of components that are being integrated into the die along with the processor(s). Table 1.1 shows the number of different SKUs associated with Intel® Core™ desktop processor family (Source: `http://ark.intel.com`). Each SKU provides a different configuration option to the end consumer to build a platform while simultaneously ensuring that each manufactured die can be sold as belonging to one of these SKUs.

Silicon units need to be tested before they can be shipped to the market. This is done using

Table 1.1: SKUs associated with Intel® Core™ desktop processor family

Intel® Core™ i7 Desktop Processor Extreme Edition	4
Intel® Core™ i7 Desktop Processor	15
Intel® Core™ i5 Desktop Processor	17
Intel® Core™ i3 Desktop Processor	7

testers or ATEs (Automated Test Equipment). A test program is embedded software that runs on top of the tester operating system. It interacts with the tester software API functions in order to provide stimulus to the Silicon unit and obtain its corresponding response. Note that this definition of a test program differs and is bigger in scope than one that refers to a sequence of instructions used to test processors. The primary objectives of a test program are:

- Determine whether a Silicon unit is defective or faulty.
- For a defect-free unit, determine the SKU that the Silicon unit belongs to.
- For a defective unit, determine the failure type for faster debug and diagnosis.

In order to meet the above objectives, a test program consists of a large number of tests that are connected in a complex fashion. The tests are diverse in nature such as DC tests, structural (scan-based) tests, functional tests, I/O tests, power-based tests, thermal tests etc. They are executed in a specific sequence in order to meet the above objectives in as short a time as possible (longer test times have an adverse impact on time-to-market). The number of lines of code needed to implement such a test program can easily run into 100,000s to a million. With the increasing complexity of the tester programs that are used to test manufactured chips, checking for their correctness has become extremely difficult. Given the nature and complexity of the test program, the threshold for a bug to creep into a test program is quite high. A test program bug can have disastrous consequences, such as:

- Defective unit being shipped to market resulting in customer returns.
- A good unit being binned to a wrong SKU which also may result in customer returns and bad end user experience.
- A good unit being discarded as a defective unit affecting yield.
- Long test times which in turn increase time-to-market and test costs.

The frequently used approach to validate a test program is to run a large number of Silicon units on ATEs. The SKUs and failure types identified by the test program are verified using statistical techniques that predict the distribution of the Silicon units across different SKUs. Any outliers indicate a bug in the test program. Such an approach is associated with quite a few drawbacks:

- Capital and run-time costs associated with a tester are prohibitively high making validation an expensive task.
- A good sample set of Silicon units to exercise different corners of a test program is hard to find early in the product development cycle.
- Statistical verification usually requires a large sample set and does not guarantee that the test program is bug-free.
- Using Silicon units to validate a test program sometimes turns the validation process into a chicken or egg problem.

Automatic software validation approaches are not tester dependent and hence, not associated with any of the above drawbacks. Early feedback regarding test program quality helps in reducing the length and frequency of test program development cycles and hence, directly impact the time-to-market of a product. Thus it is extremely important to efficiently test the programs running on the ATE.

The test programs running on ATE consists of different tests that are executed sequentially depending on the outcome of previous tests. Hence, these test programs can be modeled as loop-free, stateless but control intensive programs. The symbolic execution based strategies are very effective in generating test inputs that cover all portions of the control intensive programs. Hence they can be efficiently used to generate intelligent test inputs for the ATE programs. This helps in comprehensive testing of the test programs in an off-line environment.

Achieving complete branch coverage of the program effectively means generating inputs that will exercise all the tests at least once. These inputs cover all the corner cases and help in comprehensively testing the ATE programs. However, the previous strategies symbolic execution-based strategies have some drawbacks which make them inefficient for validation of these test programs. The aim of this thesis is to develop a strategy that can achieve complete branch coverage in a short time and which is scalable to large test programs.

1.2 Contributions

In this thesis, we propose a 2-Phased strategy for achieving high branch coverage for software quickly using symbolic execution based test generation. The Phase 1 of the strategy aims at covering as many branches as possible by targeting the shortest path to each branch node. Because each branch node is targeted by only one path, Phase 1 covers more number of branches in a short duration. Also the shorter path prefixes make constraint solving easier and thus result in fast test generation. For the branches that are unexplored in Phase 1 due to infeasible prefixes, Phase 2 is applied. Phase 2 is begins with a previously proposed

DFS-based reachability-guided search strategy. This strategy uses the reachability graph of the program and tries to find paths to unexplored branches. In both phases a simple light-weight conflict driven learning is used to avoid SMT solver calls for path prefixes that have earlier encountered conflicting conditions. We also present a new intelligent conflict driven learning for Phase 2 which intelligently skips many SMT solver calls by checking if a feasible path exists from the current path to the unexplored branch. In so doing, we can avoid all the conflicting conditions in the database.

1.3 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2 describes the concepts related symbolic execution-based test generation. The previous work related to this research is also covered in this chapter.

Chapter 3 describes the Phase 1 of our 2-Phased strategy.

Chapter 4 explains the need for intelligent conflict driven learning for Phase 2 followed by the Phase 2 algorithm of our 2-Phased strategy.

Chapter 5 shows the effectiveness of the 2-Phase strategy by presenting some experimental results.

Chapter 6 concludes the thesis and provides ideas for future work.

Chapter 2

Background

Testing of software accounts for about 50% of the development cost [1]. According to a study by the National Institute of Standards and Technology [2], software bugs or errors are so prevalent that they cost the US economy an estimated $59.5 billion annually. The study also found that more than a third of these costs could be saved by improving the testing infrastructure.

For effective testing of a software program, the desire is that all possible behaviors of the program can be exercised by the test suite. This is not achievable by applying random inputs or manual test generation. Random inputs tend to exercise the same execution paths while hard to reach paths remain untested.

Unit Testing is a commonly used technique for testing of functional behavior of the program. In unit testing, the program is decomposed into units where each unit is a function or a collection of functions. Each unit is tested independently by applying random inputs or manually generated test inputs. But manual specification of such test inputs is labor intensive and fails to cover corner cases [9].

Automation of the testing process can result in both reduced development costs and an increase in confidence in the quality of the software [3]. Hence, various techniques [3–5] have been developed for automated test generation of test inputs. In [4], data dependence analysis is used to guide test data generation. Data dependence analysis automatically identifies statements that affect the execution of some selected statement and this information is used to guide the search process. [5] applies a cost-function to every possible program input to measure how good the input is. An input selection technique is presented in [3] which selects a small subset from a large set of test inputs that is likely to reveal faults in the software under test.

In recent years, symbolic and concolic execution based techniques have gained popularity for dynamic test generation of test inputs.

2.1 Symbolic execution-based test generation

In Symbolic execution [10], the program is executed using symbolic variables instead of concrete values. The program is first instrumented using an intermediate language such as CIL [11]. When the instrumented code is executed with symbolic inputs, it generates a trace of the path followed by the program. This trace consists of the symbolic expressions of the conditions encountered along the path. The conjunction of the symbolic expressions along the path forms a path constraint. By negating one of the symbolic expressions along the current path, the path constraint of a target path is constructed. This path constraint is then converted into a formula that is solved by an underlying constraint solver, such as a Satisfiability Modulo Theory (SMT) solver [12], [13]. If the SMT solver returns a set of assignments that satisfies the formula, then the test input can be extracted from the assignment for which the formulated path can be exercised. If the SMT solver does not

Table 2.1: Example of Symbolic Execution

Line No	Original program	Symbolic value of x	Symbolic value of y	Conditional Expression	Branch taken
1	int f(int x, int y)	x	y		
	{				
2	if(x≥0)			$x \geq 0$	*true*
3	x=x+2;	$x+2$	y		
4	if(y<0)			$y<0$	*false*
5	y=y-3;				
6	if(x!=y)			$x+2!=y$	*true*
7	//some function call				
8	else				
9	//error				
	}				

find a satisfying assignment, then the path is declared infeasible, i.e., the path cannot be exercised by any valid input. By negating one of the conditional expressions in the previous path, a new path can be formed, which is again given to the SMT solver. Thus the test generation continues by finding test inputs for every possible path in the code.

Table 2.1 shows an example of symbolic execution. The program was first executed with concrete values of **x=0** and **y=0**. The second column in the table is the original program to be tested. The next two columns represent the symbolic memory map for input variables x and y respectively. The next column shows the symbolic conditional expression for all the conditions encountered along the current execution path of the program. The last column reports whether the true or false branch was taken during the concrete execution.

It can be observed from the code that when the code was executed with concrete values x=0 and y=0, the path taken by the program would be along line numbers $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7$. Before concrete execution, the program is instrumented to create a log of the path followed by the program. Now the program is run with symbolic inputs, say x

and y respectively, along the same path. During symbolic execution, the values of symbolic variables are updated according to the instructions encountered. For example, when the instruction x=x+2 is executed symbolically the symbolic value of x is updated from x to $x+2$. Similarly symbolic expressions are formed for all the conditions encountered along the path and depending upon whether the true or false branch was taken during the concrete execution, the conditional expression is evaluated to true or false. For example, during the concrete run the condition (x≥0) evaluated to true. Hence the symbolic expression for this branch is formed as *(x≥0=true)*. This expression is termed as the ***branch constraint***. The collection of all branch constraints along the symbolically executed path form the ***path constraint***. In the running example the path constraint formed after symbolic execution will be *(x≥0=true)∧(y<0=false)∧(x+2!=y=true)*

The set of all values of the input variables that satisfy this path constraint will force the program execution through the same path. In order to explore a new path we negate one of the conditional expressions in the path constraint. For example, if the last branch constraint is reversed then the path constraint to be solved by the constraint solver will be *(x≥0=true)∧(y<0=false)∧(x+2!=y=false)*. The constraint solver returns input values that can satisfy the formula (for example, x=2,y=4). When the program is executed with these values the error on line number 9 is exposed. If the constraint solver declares the formula to be unsatisfiable, then it means that the path is infeasible, i.e., there exists no input values that can force the program execution along the path in consideration. Thus, symbolic execution can formally prove whether some error condition like the assertion failure can be reached or not.

In order to systematically explore the path space of the program most of the current tools first perform a static analysis of the program to be tested and build a static reachability graph or control flow graph(CFG) of the program. In this graph each condition is given a unique ID. The graph then represents the next condition that will be encountered if true or false branch of the condition is executed. This CFG is used to maintain a record of code coverage at any given instance during test generation. Many strategies exist that uses the CFG to determine which branch constraint from the path constraint is to be reversed in order to explore a new path. Figure 2.1 shows the reachability or Control Flow Graph of the program in Table 2.1. Here the condition ID is same as the line number in the table.

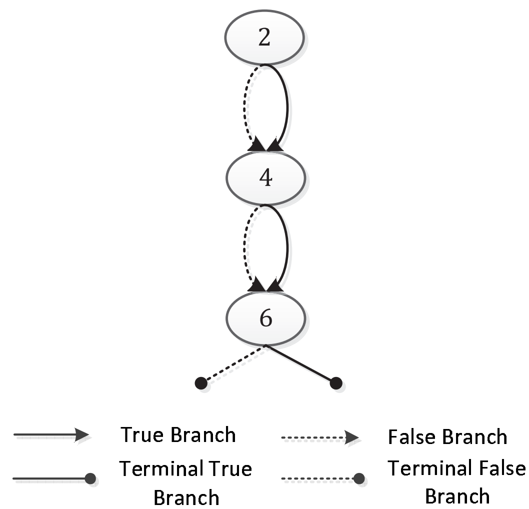

Figure 2.1: Example of a Control Flow Graph

2.1.1 Program Instrumentation

The program to be tested is first instrumented using an intermediate language. Program instrumentation helps in logging details of the path followed during concrete execution which are used during symbolic execution. A good intermediate language should be simple to analyze, close to the source and able to handle real-world code.

CIL [11] is an OCaml application for parsing and analyzing C code. The C programming language is well-known for its flexibility in dealing with low-level constructs. Unfortunately, it is also well-known for being difficult to understand and analyze, both by humans and by automated tools. Compared to C, CIL has fewer constructs. It parses through the program and converts complicated constructs in C into simpler ones. All looping constructs are reduced to a single form. It replaces break and continue statements with goto-label. It converts multiple conditions into nested conditions. It also adds explicit return statements to all functions. CIL moves all type declarations to the beginning of the program and gives them global scope. These simplifications reduce the number of cases that must be considered when manipulating a C program, making it more amenable to analysis and transformation. In addition to this we can use CIL to add statement IDs/condition IDs and insert function calls throughout the program that can generate a trace of the execution.

CIL attempts to both distill the C language constructs into a few constructs with precise interpretation and also stay fairly close to the high-level structure of the code so that the results of the source-to-source transformations bear sufficient resemblance to the source code. This makes it possible to map back the conclusions about the CIL program to the original C program. CIL syntax has three basic concepts: expressions, instructions and statements. Expressions represent functional computation, without side-effects or control flow. Instructions express side effects, including function calls, but have no local control flow. Statements capture local control flow.

Example 1: Conversion of multiple conditions to nested conditions and addition of explicit return statement.

Original Code

```
#include <stdio.h>


int main() {
  int x,y;
  if(x==0 && y==1)
  printf("Hello world\n");
```

```
}
```

CIL output

```
/* Generated by CIL v. 1.3.7 */


#line 339 "/usr/include/stdio.h"
extern int printf(char const   * __restrict  __format  , ...) ;
#line 3 "test.c"
int main(void)
{ int x ;
  int y ;
  {
#line 5
  if (x == 0) {
#line 5
    if (y == 1) {
#line 6
      printf((char const   * __restrict  )"Hello world\n");
    } else {
    }
  } else {
  }
#line 7
  return (0);
}
}
```

Example 2: Conversion of different looping constructs to **while(1)** loop

Original Code

```
include <stdio.h>
```

```
int main() {
  int x,y;
  for(x=0;x<5;x++)
  printf("%d",x);

  while(y<x)
  {
    printf("%d",y);
    y++;
  }
}
```

CIL output

```
/* Generated by CIL v. 1.3.7 */
/* print_CIL_Input is true */

#line 339 "/usr/include/stdio.h"
extern int printf(char const   * __restrict  __format  , ...) ;
#line 3 "test2.c"
int main(void)
{ int x ;
  int y ;
  {
#line 5
  x = 0;
#line 5
  while (1) {
#line 5
    if (x < 5) {
    } else {
```

```
#line 5
        break;
      }
#line 6
      printf((char const   * __restrict  )"%d", x);
#line 5
      x = x + 1;
    }
#line 8
    while (1) {
#line 8
      if (y < x) {
      } else {
#line 8
        break;
      }
#line 10
      printf((char const   * __restrict  )"%d", y);
#line 11
      y = y + 1;
    }
#line 13
    return (0);
}
}
```

When the instrumented program is executed, it generates a trace of the path followed by the program. This trace is then used to execute the program with symbolic inputs. The trace generated also includes information about the conditional expressions encountered along the path and whether they were evaluated to true or false. At the end of the symbolic execution, a path constraint is obtained which is a logical formula in terms of symbolic inputs. All the concrete input values that satisfy this formula will force the program through the same path. Depending on the search strategy some conditional expression in the path constraint is negated and the formula is given to a constraint solver. The input values returned by the constraint solver will now force the program execution through the new selected path.

2.2 Concolic execution

Concolic execution [9] is an improvement over symbolic execution. Concolic stands for co-operative concrete and symbolic execution of the program. The program is instrumented to add a function call per statement of the program. The original code of the program performs the concrete execution and the inserted function calls perform the symbolic execution. Concrete values are used to simplify complex constraints which the underlying constraint solver cannot handle.

In CUTE [9], the code is first instrumented to add function calls per statement. Then the instrumented code is run repeatedly. The logical input map I is used to generate concrete memory input graphs for the program and two symbolic states, one for pointer values and one for primitive values. The code is run concretely on the concrete input graph and symbolically on the symbolic states, collecting constraints (in terms of the symbolic variables in the symbolic state) that characterize the set of inputs that would (likely) take the same execution path as the current execution path. According to the path exploration strategy one of the collected constraints is negated. The resulting constraint system is solved to obtain a new logical input map I' that is similar to I but (likely) leads the execution through a different path. Thus the program is executed concretely and symbolically at the same time. On the other hand, in symbolic execution, the concrete execution generates a trace which is then used for symbolic execution. Hence, concolic execution is usually faster then symbolic execution.

2.3 Satisfiability Modulo Theory (SMT) solvers

Satisfiability is one of the most fundamental problems in theoretical computer science, namely the problem of determining whether a formula expressing a constraint has a solution. The most well-known constraint satisfaction problem is propositional satisfiability SAT, where the goal is to decide whether a formula over Boolean variables, formed using logical connectives, can be made true by choosing true/false values for its variables[14]. However, Boolean logic is not expressive enough for representing many problems. Such problems are expressible in decidable fragments of first order logics (or simply theories henceforth), where the propositional variables are combined with constraints over individual variables. The problem of evaluating the satisfiability of first order formulas with respect to some background theories is called Satisfiability Modulo Theory (SMT) [13].

A theory is a set of axioms and rules of inference in which first order logic predicates are interpreted. Formally, a theory T is the set of axioms and all deducible formulas from the rules of inference (all true statements). Examples of theories typically used in SMT solvers are the theory of real numbers, the theory of integers, the theories of various data structures such as lists, arrays, bit vectors and so on. A formula F is T-satisfiable if F $\wedge$ T is satisfiable in the first order sense. If not, F is T-inconsistent, or T-unsatisfiable.[12]

SMT solvers can work at mixed level of abstraction and can reason about first order formulas involving a mixture of Boolean values, bit-vectors, linear arithmetic, uninterpreted functions, difference logic, records, tuples, etc. The path constraints generated during symbolic execution can be expressed as first order formulas in theories which the SMT solvers can handle.

For deciding satisfiability or unsatisfiability of formulas in this kind of logics, during the last few years many successively more sophisticated techniques have been developed, most of which can be classified as being eager or lazy. In the eager approaches the input formula is translated, in a single satisfiability preserving step, into a propositional CNF, which is checked by a SAT solver for satisfiability. In the lazy approaches each atom in the SMT formula is internally assigned a Boolean variable. The Boolean formula is solved by an underlying SAT solver. The assignment returned by the SAT solver is then checked by the corresponding dedicated theory solvers. If any clause is violated then additional learning clauses are added in the context by the theory solvers and the search for a satisfying model continues. This process is repeated until a model compatible with the theory is found or all possible propositional models have been explored [15].

All the latest SMT solvers are based on DPLL(T) framework. In this framework the responsibility of Boolean reasoning is given to the Davis-Putnam-Logemann-Loveland (DPLL)-based SAT solver [16, 17] which, in turn, interacts with a solver for theory T through a well-defined interface. The theory solver need only worry about checking the feasibility of conjunctions of theory predicates passed on to it from the SAT solver as it explores the Boolean search space of the formula. For this integration to work well, however, the theory solver must be able to participate in propagation and conflict analysis, i.e., it must be able to infer new facts from already established facts, as well as to supply succinct explanations of infeasibility when theory conflicts arise. In other words, the theory solver must be incremental and backtrackable [18]. If the formula is satisfiable then the SMT solver returns a model i.e., values of the variables that satisfy the formula.

Some of the currently popular SMT solvers are Yices [19], MathSAT 5 [20], CVC3 [21]. In our research we have used the Yices SMT solver.

2.4 Previous work

Symbolic execution was introduced in late 70's. SELECT [7] and EFFIGY [8] were the first systems that demonstrated symbolic execution. SELECT was an experimental system for assisting in formal systematic debugging of programs. It performed symbolic execution of paths of programs written in C. Effigy was a similar tool that handled programs written in PL/I style programming language.

Many tools like DART [22], CUTE [9], CREST [23], Java PathFinder [24], have been developed in recent years that use symbolic execution for automated generation of test inputs. DART [22] presented a technique for automatic generation of new test inputs to systematically direct the execution such that the program sweeps through all its feasible execution paths. DART could handle only integer constraints and used random inputs when the pointer constraints were encountered. Because of this some paths of the program were missed. CUTE [9] further extended this technique to a broad class of sequential programs by providing a method for representing and solving approximate pointer constraints to generate test inputs. [25] used *best-first search* heuristic to drive the execution along *interesting* execution paths, e.g., that cover unexplored statements.

DART and CUTE are based on the conventional depth-first search (DFS) strategy for path exploration. In a DFS strategy, after each execution, the branch constraint of the last branch in the path constraint whose counter branch is unexplored from the current path prefix is reversed to explore a new path. Thus the DFS based symbolic execution ends after sweeping though all possible paths in the program. However, such a path exploration strategy is not scalable because the number of paths increases exponentially with the number of conditions in the code. For large programs the number of paths to be explored is so large that both symbolic and concolic execution based techniques are able to explore only small parts of the program state space in a reasonable amount of time. Hence, in [26] an algorithm for *hybrid concolic testing* was presented that interleaved random testing and concolic testing to achieve deep and wide exploration of the program state space. Hybrid concolic testing starts by performing random testing. When random testing fails to explore new points after a predetermined number of steps, the algorithm automatically switches to concolic execution from current program state to perform an exhaustive bounded DFS for an uncovered coverage point.

CREST [23] is an automatic test generation tool for C. It has a number of heuristic search strategies for tackling the path explosion problem.

- Bounded Depth-First Search - In the conventional DFS the search ends when all feasible paths have been explored. But in Bounded DFS for a bound $d > 0$, the search is restricted to the first d feasible branches along any path.
- Uniform Random Search - In this strategy the program is executed along random paths instead of random inputs. Hence it avoids the problem in random testing that often many inputs are used that lead to same execution paths.
- Random Branch Search - In this strategy a branch along current path is randomly chosen and the execution is forced to not take that branch. Thus, this strategy takes a random walk through the path space.
- Control-Flow graph Directed Strategy - This strategy uses the static structure of the program under test to guide the dynamic search of the program's path space. A static control flow graph (CFG) of the program under test is built. Then, the distances of all

the uncovered branches from the branches along the current path are determined after each execution and the branch with the minimum distance to an uncovered branch is negated to explore a new path that will bring us closer to exploring an uncovered branch.

A DFS based reachability guided strategy was proposed in [27] which performs branch coverage using the static reachability graph of a program. Since our research is an based on this strategy it is necessary to analyze at this strategy in detail.

2.4.1 DFS-based reachability-guided strategy

A DFS based reachability guided strategy proposed in [27] performs branch coverage using the static reachability graph of a program. In a DFS strategy, after each execution, the branch constraint of the last branch in the path constraint whose counter branch is unexplored from the current path prefix is reversed to explore a new path. This strategy makes use of a reachability graph of the given program, and after each execution, starting from the terminal node, it checks if the counter edge can reach any unexplored branch. The counter edge is explored only if it leads to any important conditional that has not yet been visited, or if it itself is an important condition. This is achieved by performing a DFS on the sub-tree at the node of the counter edge with the help of the reachability graph. If no new important branches can be reached from the counter edge then the search procedure backtracks to the previous node in the CFG and this process continues until all the branches are covered. The complete algorithm followed by this strategy is shown in Algorithm 2.1 [27].

This strategy was able to achieve complete branch coverage in a reduced execution time, as compared to the earlier strategies. However, the performance of such an approach can be limited by focusing too much on trying to reach the unexplored branches from the current

Algorithm 2.1 DFS-based reachability-guided search strategy

```
Path ← ∅, Tests ← ∅
CurrNode ← initial node
append CurrNode to Path
while Path is not empty do
  if not covered any branches of CurrNode then
    transition ← false branch of CurrNode
    if transition can reach any unvisited important conditionals then
      if transition does not lead to terminal node then
        append transition to Path
      else
        if path constraint of Path has solution then
          record Tests
        else
          record Path as infeasible
        end if
      end if
    end if
  else if covered only one branch of CurrNode then
    /*symmetric case for true branch*/
  else
    remove CurrNode from Path
    CurrNode ← previous node in Path /*backtrack*/
  end if
end while
```

execution path as will be explained in next subsection. In other words, there may exist better paths to reach the unexplored branches.

2.4.2 Disadvantages of the DFS based reachability-guided strategy

Even though the DFS based reachability-guided strategy achieves complete branch coverage in a reduced execution time, it has some disadvantages which degrade its performance, explained below:

1. The DFS based strategy tries to explore every node, n, in the reachability graph and the sub-tree under n through the first path that reaches n. Thus, it may often fail to take into account the fact that multiple incoming edges (and paths) may exist to a given node, and hence a target node can be accessed through alternative paths that may be superior with regard to test generation time and complexity. Because of the lack of this consideration, the reachability-guided strategy may sometimes end up exploring a target node through longer paths even though a shorter path exists to that node.

   Consider the reachability graph shown in Fig. 2.2. Here the initial path taken is the bold-faced path as shown in Fig. 2.2a. Since $(n3, F)$ is a terminal edge, the DFS based strategy checks if $(n3, T)$ leads to any unexplored branch. As $n2$ and nodes beneath $n2$ have not yet been explored, it will now explore it though the path in Fig. 2.2b. Now for node $n2$ and nodes beneath $n2$, there exists a shorter path prefix through $(n0, T)$. The DFS based technique does not take this shorter path into account when extending the test generation from the current path.

2. Since the strategy is reachability based, it sometimes can get stuck in a subspace of the reachability graph. This occurs when a node has many paths but very few (or none)

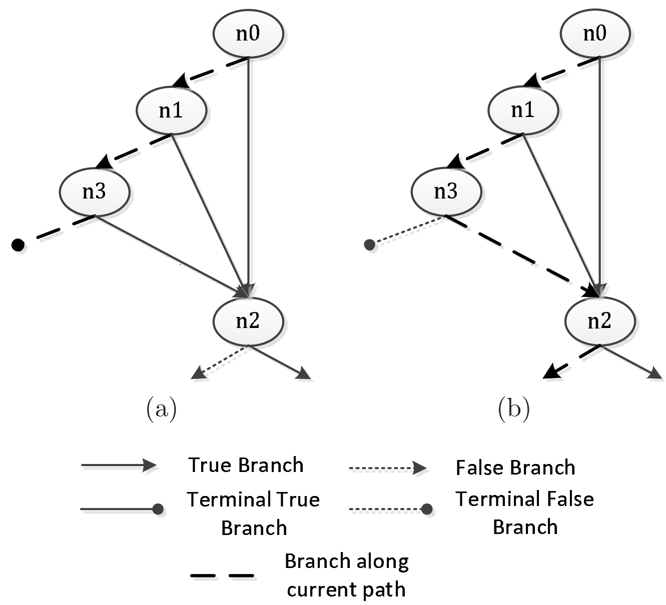

Figure 2.2: A reachability graph where the DFS based reachability guided strategy explores a node through a longer path even though shorter paths exist

of them are feasible. Consider the reachability graph in Fig. 2.3. Here the initial path taken is $(n0, F) \rightarrow (n1, T) \rightarrow (n4, F) \rightarrow (n5, F) \rightarrow (n6, T) \rightarrow (n7, F)$. After reversing the last condition according to DFS we try to explore the path $(n0, F) \rightarrow (n1, T) \rightarrow (n4, F) \rightarrow (n5, F) \rightarrow (n6, T) \rightarrow (n7, T)$ which turns out to be infeasible. Now, we backtrack and continue with the path exploration. It can be seen that the node $n7$ is reachable from the nodes $n0$ to $n6$. So with the DFS based search, node $n7$ is attempted from all these nodes. For a path like $(n0, F) \rightarrow (n1, F) \rightarrow (n2, F) \rightarrow (n3, F)$, node $n7$ is reachable from $(n3, T)$, so this path is considered just to explore $(n7, T)$. And it might take multiple hops to reach $(n7, T)$. By intelligent observation it can be seen that in the worst case around 15 SMT solver calls can be made just to attempt to explore $(n7, T)$ while the nodes beneath $(n0, T)$ remain unexplored for a long time. Thus, the branch exploration can be confined to a subspace and this reduces the efficiency of the DFS based reachability-guided strategy and makes it less scalable for larger programs.

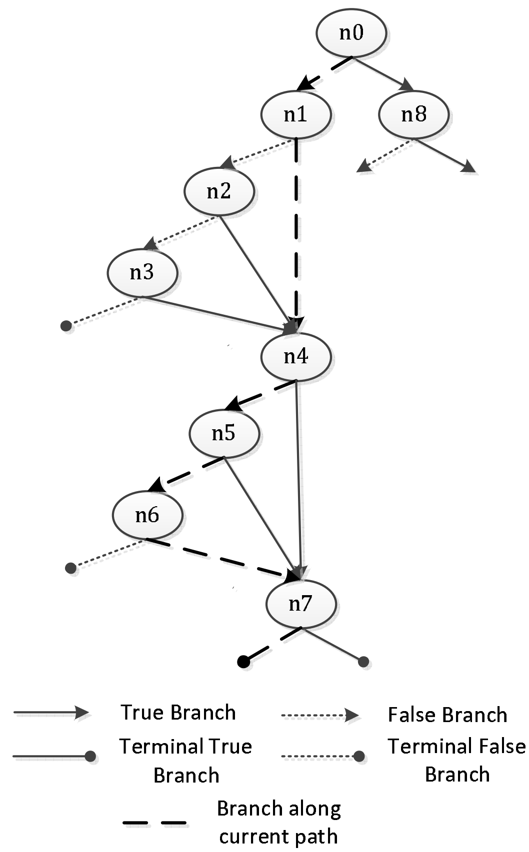

Figure 2.3: A reachability graph where a DFS based strategy gets confined in a subspace for a long time

3. While trying to explore an unexplored branch, the DFS based strategy may often find many paths infeasible due to same set of conflicting conditions. For example, in the previous example the branch $(n7, T)$ was tried to be explored through path $(n0, F) \rightarrow (n1, T) \rightarrow (n4, F) \rightarrow (n5, F) \rightarrow (n6, T) \rightarrow (n7, T)$ which was infeasible. If the conflicting branches making this path infeasible were $\{(n0, F), (n7, T)\}$ then all the paths that start with branch $(n0, F)$ and reach $(n7, T)$ will be infeasible because of the same set of conflicting branches.

The aim of this research is to overcome these disadvantages and develop a strategy which achieves very high branch coverage quickly and which is not bogged down by regions that consume excessive search due to presence of infeasible paths. In this thesis, we address the aforementioned challenges by favoring short paths first and then cleverly targeting the remaining unexplored branches. To do so, we propose a 2-phase strategy that explores each branch through the shortest path to that branch in Phase 1, and then explores the branches that were infeasible in Phase 1 through other available paths. To reduce the computational cost further, in Phase 1, a basic conflict driven learning is used to skip all the paths that may have any of the earlier encountered conflicting conditions, while in Phase 2, a more intelligent conflict driven learning is used to skip regions that do not have a feasible path to any unexplored branch. This results in considerable reduction in unnecessary SMT solver calls and significant speedup can be achieved [28].

Chapter 3

Phase 1

The 2-phased strategy makes use of the reachability graph of the program and tries to explore every node only through the shortest path to that node in Phase 1.

3.1 The Strategy for Phase 1

In Phase 1, we first perform a quick breadth-first search (BFS) on the reachability graph as a preprocessing step. We mark those input edges to any node, n, which do not form the shortest path prefixes to n and the nodes beneath it.

After the edges are marked, there is exactly one path from root node to every node in the reachability graph such that the path does not contain any of the marked edges. This is the shortest path from the root node to that node. This information is used to make sure that each node and the sub-tree under it are explored only through the shortest possible path in Phase 1.

We start with the path obtained after executing the code with an initial test input (could be random). After execution of the initial input, we check if the path taken contains any of the edges that were marked earlier. If so, we truncate the path till a marked edge closest to the root of the graph, and we record the coverage only till that marked edge. We now find the next path by flipping the direction of the last node if it has not been already covered. Otherwise, we backtrack to the previous node. A detailed algorithm of our approach is given in Algorithm 3.1.

Algorithm 3.1 Algorithm for Phase 1

```
not_to_be_visited_edges ← BFS of reachability graph
Path ← execute the code with initial inputs
if Path has any not_to_be_visited_edges then
    Path ← truncate Path up to the first edge in not_to_be_visited_edges
end if
record the coverage for Path
while Path is not empty do
    Path ← drop all trailing edges with explored counter edges
    if Path is not empty then
        Path ← reverse the last branch
        if path constraint of Path has solution then
            record test_inputs
            Path ← execute code for test_inputs
            if Path has any not_to_be_visited_edges then
                Path ← truncate Path up to the first edge in not_to_be_visited_edges
            end if
            record the coverage for Path
        end if
    end if
end while
```

An example of the procedure followed in Phase 1 is shown in the Fig. 3.1. Here, a portion of a reachability graph is shown (nodes below $n6$ are not shown). After performing the BFS on the reachability graph the edges that are marked as not to be explored are $\{(n2, T), (n3, T), (n3, F), (n5, T)\}$. These are marked because they are not a part of shortest path to any of nodes.

The initial inputs take the path $(n0, F) \rightarrow (n2, F) \rightarrow (n3, F) \rightarrow (n5, F)$ as shown in Fig.

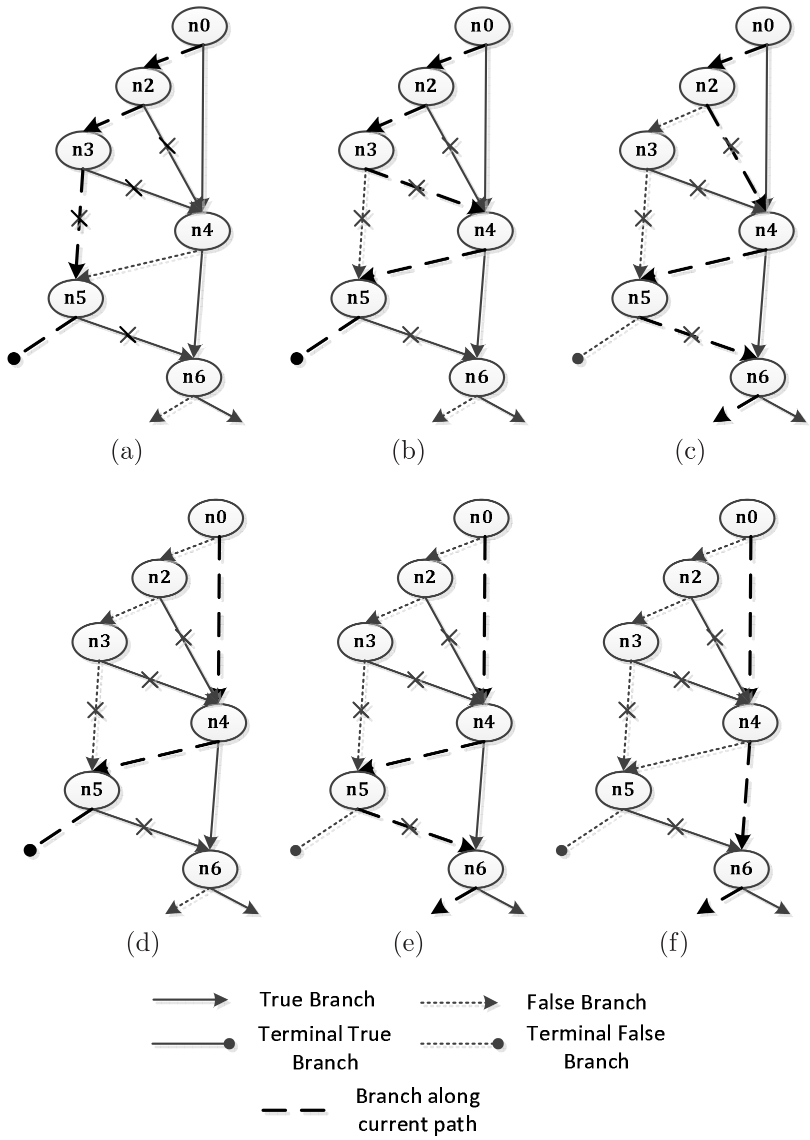

Figure 3.1: An example of procedure followed in Phase 1

3.1a. The edge $(n3, F)$ was marked as not to be explored for $n5$ and nodes beneath $n5$. Hence, we truncate the path till $(n3, F)$ and then record the coverage of the truncated path. Then we backtrack and explore the other edge of $n3$. The path taken in this case is $(n0, F) \rightarrow (n2, F) \rightarrow (n3, T) \rightarrow (n4, F) \rightarrow (n5, F)$ as shown in Fig. 3.1b. Again $(n3, T)$ is a marked edge. We truncate the path up to $(n3, T)$ and record the coverage. Since both the edges of $n3$ have been explored we backtrack and explore the other edge of $n2$ and take path $(n0, F) \rightarrow (n2, T) \rightarrow (n4, F) \rightarrow (n5, T) \rightarrow (n6, F)$ as shown in Fig. 3.1c. This path contains two marked edges $(n2, T)$ and $(n5, T)$. We truncate the path till $(n2, T)$ because it is the edge that is closest to the root node $n0$. We now backtrack and explore other edge of $n0$ and take path The path taken is $(n0, T) \rightarrow (n4, F) \rightarrow (n5, F)$ as shown in Fig. 3.1d. Now, $(n0, T)$ is the shortest path prefix for node $n4$. So, all the nodes beneath $n4$ will be explored through this path prefix. Similarly $(n0, T) \rightarrow (n4, F)$ is the shortest path for $n5$. Since this path does not have any marked edge, we record the coverage of the entire path and then explore the other edge of $n5$. The path taken is $(n0, T) \rightarrow (n4, F) \rightarrow (n5, T) \rightarrow (n6, F)$ as shown in Fig. 3.1e. This path has a marked edge $(n5, T)$. So we truncate the path till this edge and we backtrack to explore other edge of $n4$. Now the path taken is $(n0, T) \rightarrow (n4, T) \rightarrow (n6, F)$ as shown in Fig. 3.1f. This path does not have any marked edge which means this is the shortest path to node $n6$. We continue to explore the nodes beneath $n6$ in a similar way by using the smallest path prefix to each node.

3.1.1 Lower coverage reported

In the Phase 1 algorithm explained before, at every step we check if the path contains any marked edge. If it does then we truncate the path till the marked edge closest to the root and then record the coverage of the truncated path. This results in reporting of lower coverage than the actual coverage that is achieved because some branches that were covered by the path are not reported as covered.

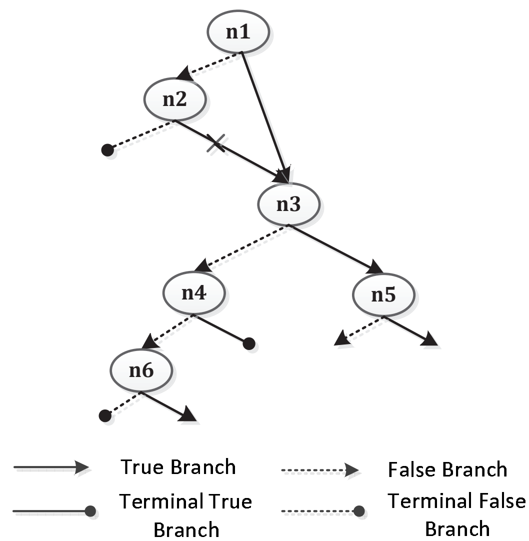

Figure 3.2: A reachability graph showing the need to report low coverage in Phase 1

The reason for choosing to report the lower coverage can be explained with the following example. In the reachability graph shown in Figure 3.2, $(n2, T)$ is the only branch marked as not to be explored after the BFS on the reachability graph. Let the path taken by initial inputs be $(n1, F) \rightarrow (n2, T) \rightarrow (n3, F) \rightarrow (n4, F) \rightarrow (n6, F))$. This path contains the marked edge $(n2, T)$. Suppose we report the coverage of the entire path and then truncate the path to $(n1, F) \rightarrow (n2, T))$. According to the Phase 1 algorithm the next path taken is $(n1, F) \rightarrow (n2, F)$. This path has no marked edge so we record the coverage of the entire path. No we backtrack to explore the branch $(n1, T)$. Let the path taken after executing the inputs generated for this path prefix be $(n1, T) \rightarrow (n3, F) \rightarrow (n4, T)$. Since this path does not contain the marked edge, we record the coverage of entire path. According to the Phase 1 algorithm we check if the counter edge of the last branch i.e. $(n4, F)$ is covered or not. This branch was reported as covered by the first path. So we backtrack. Here we can see that the branch $(n6, T)$ and the other nodes below this branch were to be explored with the path prefix $(n1, T) \rightarrow (n3, F) \rightarrow (n4, F)$. But since we backtracked without exploring $(n4, F)$ through the current path prefix, these branches will remain unexplored in Phase 1. To prevent this we can perform a DFS at node $n6$ (i.e. node at end of counter edge $(n4, F)$ to check if it reaches any unexplored node. This will make the strategy similar to the DFS based reachability guided strategy and we will face all the shortcomings of DFS based reachability guided strategy that were explained in section 2.4.2. Hence we choose to report the coverage of the truncated path and not the entire path. This results in early completion of Phase 1.

3.2 Conflict Driven Learning

As explained earlier, during path exploration some paths may be infeasible. Many of these infeasible paths may have the same set of conflicting conditions. To avoid unnecessary time spent by the SMT solver to solve a path constraint which can be predetermined to be unsatisfiable, a simple conflict-driven learning such as that introduced in [27] is sufficient for Phase 1. Every time the SMT solver returns a path constraint as infeasible we extract the unsatisfiable core which is the set of conflicting clauses that make the formula unsatisfiable.

3.2.1 UNSAT core

An unsatisfiable core can be defined as any subset of the original formula that is unsatisfiable. Consequently, there may exist many different unsatisfiable cores, with different number of clauses, for the same problem instance, such that some of these cores are subsets of others. Also, and in the worst case, the unsatisfiable core corresponds exactly to the set of original clauses [29].

An unsatisfiable core is C minimal iff the formula obtained by removing any of the clauses of C is satisfiable. A minimum unsat core is a minimal unsat core with the smallest possible cardinality[30]. The minimal unsat core is also known as minimal unsatisfiable subformula (MUS).

A naive algorithm for minimal unsatisfiable core works as follows: For every clause C in an unsatisfiable formula F, the algorithm checks if it belongs to the minimal core by invoking a propositional satisfiability (SAT) solver on F, but without clause C. Clause C does not belong to a minimal core if and only if the solver finds that F$\setminus\{$C$\}$ is unsatisfiable, in which case C is removed from F. In the end, F contains a minimal unsatisfiable core[31].

Many techniques have been proposed in recent years for the extraction of unsat core [29–34]. SMT solvers CVC3 and MathSAT 5 can compute unsatisfiable cores as a byproduct of the generation of proofs, in a way similar to that in [34]. Yices instead uses the following technique: a selector variable is introduced for each original clause, which is forced to false before starting the search. In this way, when a conflict at decision level zero is found, the conflict clause contains only selector variables, and the unsat core returned is the union of the clauses whose selectors appear in such conflict clause[30].

In our context the formula to be solved by the SMT solver is the new path prefix. In the formula each clause represents a branch i.e. true or false of a conditional expression of the program. The unsatisfiable core consists of the set of branches that made the path infeasible.

For example, consider the path prefix $(x > 0 = false) \wedge (x > z = true) \wedge (z > x = true) \wedge (z == 3))$. It can be observed that this formula is unsatisfiable i.e. this path is infeasible. The reason for infeasibility is that the two clauses $(x > z = true)$ and $(z > x = true)$ cannot be satisfied together. These two clauses are reported by the SMT solver as the unsatisfiable core. However, it must be noted that the unsatisfiable core reported by any SMT solver is not guaranteed to be minimal. If the same set of branches are present in any other path then that path will also be infeasible.

In Yices, each branch constraint is given a different ID. When the path constraint is unsatisfiable, Yices returns the IDs of the branch constraints in the unsat core. These are the conflicting branches in the path that make the path infeasible. Here each branch is a true or false direction of a condition in the code being tested. We then store these conflicting branches in a database. Henceforth, we will use the term unsat constraint for each set of conflicting branches and an entire list of such unsat constraint as unsat constraint list. Before calling the SMT solver, we check if the path whose formula is to be solved contains any of the unsat constraint already present in the database. If so, we assume the path to be infeasible and skip the solver call and move on to explore the next path. This conflict driven learning works only if the program is stateless. The stateless nature of the program guarantees that the value of variables do not change and hence are path independent. Hence, if the same set of conflicting conditions are encountered by some other path then that path is also infeasible. This conflict driven learning saves considerable amount of time because for long paths with large number of branches the solver call can be expensive.

3.2.2 Advantages of Proposed Phase 1

The advantages of the proposed Phase 1 are listed below.

1. Phase 1 quickly explores every node through the shortest path to the node, thereby reducing the time needed to explore a large portion of the nodes.

2. As Phase 1 covers only the shortest path to every node in the reachability graph, this strategy is scalable because the number of paths explored will increase linearly rather than exponentially with the increase in number of conditions in the code.

3. Since every node is explored through only one fixed path in Phase 1, the search will not be stuck in a specific region due to infeasible paths. If the shortest path to the node is found to be infeasible we move on to next node.

4. By using a simple low-cost conflict driven learning, unnecessary SMT solver calls are avoided.

Chapter 4

Phase 2 with Intelligent Conflict-driven Learning

If there are no infeasible paths during Phase 1, then Phase 1 will guarantee maximum possible branch coverage in the least amount of time. In that case, Phase 2 is unnecessary. However, it is possible that for some nodes the shortest path is infeasible. For those unexplored branches in Phase 1, we need a Phase 2 to explore these branches through alternative paths.

Phase 2 uses a DFS based reachability guided strategy similar to previous approaches, with an intelligent conflict driven learning. Phase 2 starts with all the information of the branches already covered by Phase 1 and aims at covering only the remaining branches. We note that since the DFS based reachability guided technique has been shown to be complete, maximum possible coverage will be achieved at end of Phase 2. However, this can be a time consuming phase since it is trying to find a feasible path to all the nodes that were not reached during Phase 1. Some of these nodes may have a large number of paths but very few of them feasible. In this phase, our main goal is to reduce the computational cost needed.

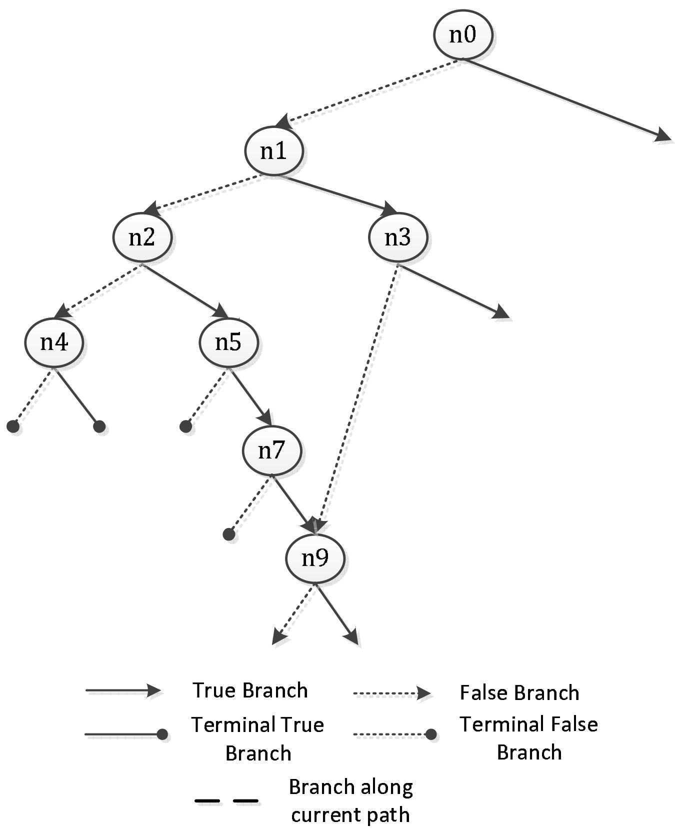

Figure 4.1: A reachability graph showing the need for an intelligent conflict driven learning in Phase 2

4.1 Need for an intelligent conflict driven learning

The simple conflict driven learning that was used in Phase 1 is not efficient for Phase 2. This can be explained with the following example.

Consider Fig. 4.1. After Phase 1 the only unexplored branch remaining is $(n9, T)$. The branch $(n9, T)$ was infeasible in Phase 1 because of the conflicting conditions $\{(n0, F), (n9, T)\}$. According to the DFS based reachability guided strategy we try to find a path to node $n9$ from the last executed path. Let last executed path be $(n0, F) \rightarrow (n1, F) \rightarrow (n2, F) \rightarrow (n4, F)$.

According to the DFS-based reachability guided strategy, we check if node at end of counter edge of the last edge in the last executed path reaches node $n9$. In this case the counter edge of the last executed path is $(n4, F)$ and it is a terminal edge. Hence it cannot reach node $n9$ so we backtrack. Now, the counter edge of $(n2, F)$ (i.e, $(n2, T)$) reaches node $n5$. So we perform a DFS on the sub-tree under node $n5$ and find that branch $(n9, T)$ is reachable. So the new path prefix chosen is $(n0, F) \rightarrow (n1, F) \rightarrow (n2, T)$. The SMT solver solves the path prefix and generates the inputs for which the code will follow this path prefix. After running the code with these inputs the path taken is $(n0, F) \rightarrow (n1, F) \rightarrow (n2, T) \rightarrow (n5, F)$. Following the same procedure the next two paths taken are $(n0, F) \rightarrow (n1, F) \rightarrow (n2, T) \rightarrow (n5, T) \rightarrow (n7, F)$ and $(n0, F) \rightarrow (n1, F) \rightarrow (n2, T) \rightarrow (n5, T) \rightarrow (n7, T) \rightarrow (n9, F)$. Now, by changing the last condition we reach the unexplored branch $(n9, T)$. The path prefix to be solved by SMT solver for this path is $(n0, F) \rightarrow (n1, F) \rightarrow (n2, T) \rightarrow (n5, T) \rightarrow (n7, T) \rightarrow (n9, T)$. However, after applying the basic conflict driven learning we find that this path consists of the conflicting conditions present in the database $\{(n0, F), (n9, T)\}$. So we avoid the solver call and look for the next available path to reach $(n9, T)$. Thus, although the basic conflict driven learning can help us avoid one solver call, the earlier three solver calls made to reach this particular path were also unnecessary. We explain this as follows. Since $(n0, F)$ is already present in the path prefix of $n2$, there is no feasible path to $(n9, T)$ from $(n2, T)$. So the conflict driven learning used in Phase 2 needs to be modified with a more intelligent conflict driven learning which not only checks if branch $(n9, T)$ is reachable from $(n2, T)$ but also checks if it is reachable by avoiding all the conflicting conditions in the database.

4.2 Phase 2 Algorithm with intelligent conflict-driven learning

The intelligent conflict driven learning in Phase 2 currently assumes the stateless property of the program. However, the learning can be extended to general programs by ensuring that the variables and the paths associated with the conflict exhibit stateless property. During path exploration in Phase 1, we choose the next path by reversing the direction of some condition in the path that was currently taken. The new path constraints are then formulated and given to the SMT solver. If the SMT solver returns the formula unsatisfiable, then the new branch with the reversed direction will be a part of the unsat core since the direction of only one condition was changed between the last path and the new path. Since each branch is explored through only one path in Phase 1, each unsat constraint generated (in Phase 1) would include an unexplored branch. In particular, the unexplored branch is the last branch in the unsat constraint. So at the end of Phase 1 the branches left to be explored are the last branches in every set of unsat constraints as well as the nodes below these branches. Now, Phase 2 only needs to target these unexplored branches. The procedure followed in Phase 2 is described in Algorithm 4.1.

In Phase 2, after each execution we check if the counter branch of the last branch is unexplored. If it is unexplored, we choose the new path by reversing the direction of last branch. If it has been explored, we perform a reachability check on the counter branch to check if the counter branch reaches any unexplored branch by avoiding all the unsat constraints in the database. The detailed procedure for checking the reachability of a branch (n, V) is explained in Algorithm 4.2. If no conclusion about the presence of a feasible path can be made at the branch (n, V), then this algorithm recursively performs the reachability check

Algorithm 4.1 Algorithm for Phase 2

```
while Path is not empty do
  if counter branch of last condition is unexplored and it is not an unreachable branch
  then
    to_explore ← true
  else
    to_explore ← reachability check (counter branch)
  end if
  if to_explore then
    reverse the direction of last condition in Path
    give the new path formula to SMT solver
    if satisfiable then
      record coverage
      Path ← execute for new inputs
    else
      get the unsat core and add a new entry to the unsat constraint list
      Path ← remove last branch from Path
    end if
  else
    Path ← remove last branch from Path
  end if
end while
```

Algorithm 4.2 Algorithm for *reachability check* of branch (n, V)

```
Let m be the node at the end of branch (n, V)
if sub-tree at node m has at least one unexplored branch then
  trim the unsat constraint list with path prefix up to (n, V)
  divide the infeasible branches into the three groups
  if reachable branch list is not empty then
    return true
  else
    if possibly reachable branch list is not empty then
      return reachability check (child branches with (n, V) added to path prefix)
    else
      return does m reach any totally unexplored node by avoiding all unreachable
      branches
    end if
  end if
end if
return false
```

on the child nodes. This process is continued till a conclusion about the presence of feasible path is reached.

We explain Algorithms 4.1 and 4.2 with the following example. Let the current path prefix for node nj be $(na, T) \rightarrow (nc, F) \rightarrow (nd, T) \rightarrow (ne, F) \rightarrow (ng, T) \rightarrow (ni, F)$. Let the unsat constraint list after Phase 1 be

$\{(nc, F),\ (nm, T)\}$

$\{(na, F),\ (nd, T),\ (ns, F)\}$

$\{(ne, F),\ (no, T),\ (nz, F)\}$

$\{(nd, T),\ (nr, T),\ (nx, F)\}$

By looking at the last branches of each of the unsat constraints, the branches (nm, T), (ns, F), (nz, F) and the sub-trees following these branches are unexplored after Phase 1 based on our previous discussion. Further, let the nodes under the sub-tree of nj be nk, nl, nm, no, np, nr, ns, nv, nw and nz.

For each of the three unsat constraints, it may either fall within or outside of the current path. We can thus examine each unsat constraint and make certain deductions. For example, consider the first unsat constraint $\{(nc, F),\ (nm, T)\}$. Since (nc, F) is on the current path, and the fact that $\{(nc, F),\ (nm, T)\}$ is unsat, we can readily deduce that (nm, T) is not feasible along the current path.

To generalize the technique, each unsat constraint can be reduced to a **trimmed unsat constraint** in the following manner. For every unsat constraint, we first check if the root branch matches the first branch in an unsat constraint. If so, we delete the branch from that unsat constraint. We then follow the same procedure for every branch along the path prefix. For above example, the three **trimmed unsat constraint** would be

$\{(nm, T)\}$

$\{(na, F),\ (nd, T),\ (ns, F)\}$

$\{(no, T),\ (nz, F)\}$

$\{(nr, T),\ (nx, F)\}$

Lemma 4.2.1. *If only one branch remains in a trimmed unsat constraint after trimming the unsat constraint list with the current path prefix, then that branch is guaranteed to not have a feasible path with the current path prefix.*

Proof. Follows from the preceding discussion on branch (nm, T). □

Hence, we classify branch (nm, T) and all such branches that satisfy Lemma 4.2.1 as an **unreachable branch** with the current path prefix.

Lemma 4.2.2. *If a trimmed unsat constraint has multiple branches and if any branch in the trimmed unsat constraint is unreachable from the ending node of the current path prefix, then that unsat constraint is said to be irrelevant with respect to the current path as it cannot be present in any path that has this current path prefix.*

Proof. For any unsat constraint to be present in a path, all the branches in that unsat constraint must be in the path. Hence, if the first branch of the trimmed unsat constraint is unreachable from the ending point of the current path prefix, then that unsat constraint cannot be present in any path starting from the current path prefix. Likewise, if the second branch of the trimmed unsat constraint is unreachable from the current path prefix, then the unsat constraint is also irrelevant, and so on for other branches in the trimmed unsat constraint. □

If the last branch of such an irrelevant unsat constraint is reachable from the ending node of current path prefix, then we term the last branch of that unsat constraint as a **reachable**

branch as the current path prefix has a path to that branch that avoids the unsat constraint in question. In our running example, consider the second trimmed unsat constraint $\{(na, F),\ (nd, T),\ (ns, F)\}$. Since node na is not in the sub-tree of nj (end point of current path prefix), this constraint is irrelevant with respect to the current path prefix. However, node ns is in the subtree of nj, hence branch (ns, F) of the second unsat constraint is a reachable branch, since there exists a path from nj to ns that avoids this second unsat constraint.

Now, if the last branch of such an irrelevant unsat constraint is not reachable from the end node of current path prefix then there is no path to that branch from the current path prefix. So we ignore this branch while exploring paths through the current path prefix. Consider the fourth trimmed unsat constraint, $\{(nr, T),\ (nx, F)\}$. Since node nx is not in subtree of nj, there is no path to node nx from the current path prefix. So we ignore the unexplored branch (nx, F) of this unsat constraint for the current path prefix.

Lemma 4.2.3. *If the trimmed unsat constraint has multiple branches and if the first branch in a trimmed unsat constraint is reachable from the ending node of the current path prefix, then there definitely exists a topological path to each of the following branches in the constraint. (Note that this topological path may or may not be sensitizable.)*

Proof. Since the unsat constraint is obtained after solving a path constraint, there is at least one path that has all the branches in the unsat constraint. So if the first branch in the trimmed unsat constraint is reachable from the ending point of the current path prefix, there definitely exists at least one topological path that can lead to all the remaining branches in the trimmed unsat constraints. □

But we cannot guarantee that the topological path will be a feasible path, because all the paths with the current path prefix leading to the last branch of the unsat constraint may have the unsat constraint in it. Hence, we term the last branch of the unsat constraint as **possibly reachable**. In the above example, since no is reachable from nj, (nz, F) (from the third trimmed unsat constraint) is a possibly reachable branch.

When dividing the infeasible branches into the above three groups we must make sure that each branch is present in only one of the three groups. In the unsat constraint list obtained after Phase 1, the infeasible branches will be unique as each branch is explored through only one path in Phase 1. But during Phase 2, we might get another unsat constraint for the same branch. Hence, we prioritize the lists in descending order of priority as *unreachable branches, possibly reachable branches* and *reachable branches.*

From Algorithm 4.2 it can be seen that the reachability check can be computationally intensive for some reachability graphs. To avoid some unnecessary reachability checks, we use the *no feasible path nodes list*. To keep the algorithm listings simple, *no feasible path nodes list* is absent in the Algorithm 4.1 and 4.2.

The **no feasible path nodes list** is a list of nodes that do not lead to any feasible paths to any unexplored branch for the current path prefix. It is used to improve the speed of the reachability check in a heavily connected graph as shown in Fig. 4.2

Here, consider the current path is $(n1, T) \rightarrow (n3, F) \rightarrow (n4, T) \rightarrow (n5, F) \rightarrow (n7, F) \rightarrow (n9, F)$. We are performing a reachability check for branch $(n9, T)$ under the path prefix up to $(n7, F)$. The reachability check algorithm first checks if there exists a feasible path in the sub-tree under the $(n10, T)$ branch with the current path prefix. If not, it will check if there is a feasible path in the sub-tree under the $(n10, F)$ branch. When the reachability check is performed on branch $(n10, T)$ it recursively covers nodes in the sub-tree, namely $n12$, $n14$, $n15$, $n17$, $n18$, $n19$, $n20$ and $n21$. Again, when the reachability check is performed on branch $(n10, F)$, it will also recursively perform the reachability check on the corresponding

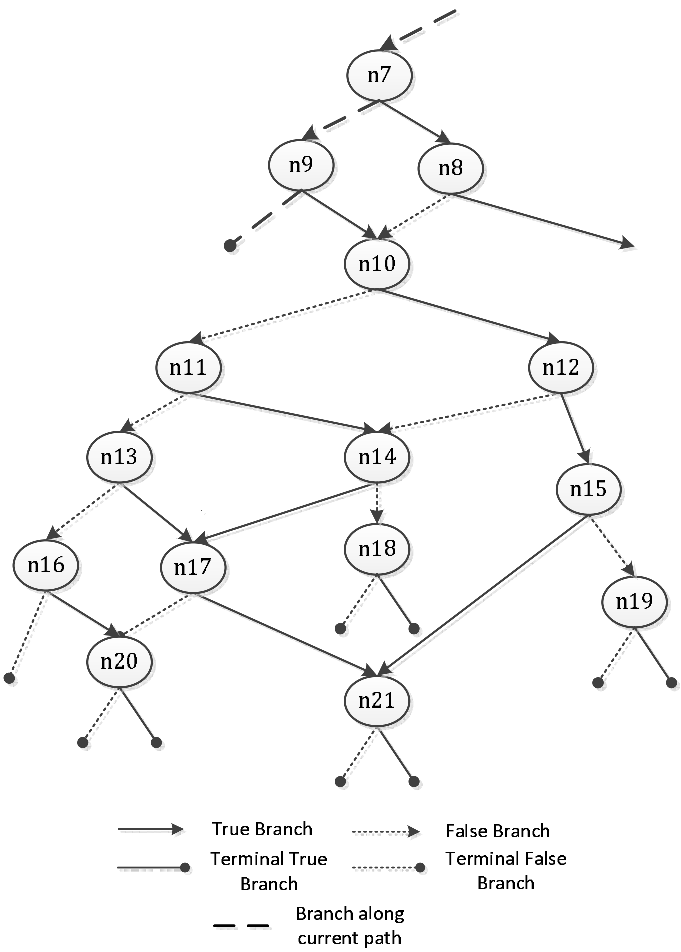

Figure 4.2: A reachability graph showing the use of *no feasible path nodes list*

sub-tree, which includes nodes $n11$, $n13$, $n14$, $n16$, $n17$, $n18$, $n20$ and $n21$. Here, we notice that the reachability check will be performed **twice** on nodes $n14$, $n17$, $n18$, $n20$ and $n21$.

Now, if the unsat constraint list trimmed with path up to $(n10, T)$ remains unchanged when branch $(n10, T)$ is removed from the path, the nodes which have returned false while recursively performing the reachability check of $(n10, T)$ will definitely remain false during reachability check of $(n10, F)$. This is because the outcome of reachability check depends only on the trimmed unsat constraint list, rather than the complete path. But if there is a change in any of the trimmed unsat constraint after removing branch $(n10, T)$, then some branch that did not have any feasible path earlier through $(n10, T)$ may now have a feasible path through $(n10, F)$. Hence, if removing branch $(n10, T)$ does not affect any of the trimmed unsat constraint, then we pass the *no feasible path nodes list* obtained after performing reachability check on branch $(n10, T)$, when performing reachability check on branch $(n10, F)$. This allows us to avoid many unnecessary reachability checks on nodes that are reachable from both $(n10, T)$ and $(n10, F)$. By using the same reasoning, we continue to hold the *no feasible path list* when we backtrack to the previous node in the current path if removing the last branch from the path does not affect the trimmed unsat constraint list. Thus by using the *no feasible path list* a considerable time is saved in the reachability check.

During the reachability check it is not sufficient just to check if any of the last branches in the unsat constraint is reachable. Once we have concluded that there are no *reachable* or *possibly reachable* branches from branch (n, V), we also need to check if branch (n, V) reaches any totally unexplored node (i.e., a node with both branches unexplored). For example, node $n12$ in Fig. 4.3 is totally unexplored because branch $(n10, T)$, which is on the shortest path to node $n12$, was infeasible in Phase 1. Hence, branches $(n12, T)$ and $(n12, F)$ will not be a part of any unsat constraint list. Now in Phase 2, while performing reachability check for some node before node $n10$, we conclude that branch $(n10, T)$ is *unreachable*. But there may still exist a path through $(n10, F) \rightarrow (n11, T)$ that can reach node $n12$ by avoiding the unsat constraint of branch $(n10, T)$. To check for such cases, once the *possibly reachable branch list* gets empty we perform a quick DFS by avoiding all the branches in the *unreachable branch list* on node m, which is at the end of branch (n, V). After avoiding all the branches in *unreachable branch list*, if node m reaches any totally unexplored node, there definitely exists a feasible path to that totally unexplored node through node m.

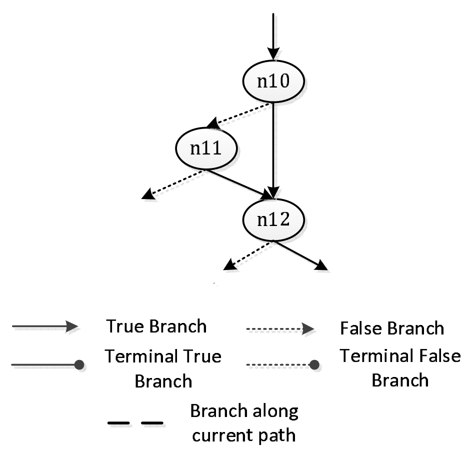

Figure 4.3: A reachability graph showing a totally unexplored node

Chapter 5

Experimental Results

We implemented the DFS based reachability guided strategy and our 2-phase strategy on Intel's internal tool. The two strategies were tested on several test cases derived from the test programs used in high volume manufacturing of Intel's semiconductor devices. As mentioned earlier, these test cases are stateless, loop free but control intensive C programs. We further compared the number of SMT solver calls needed by our 2-phase strategy with the iterations needed by the CFG directed strategy of CREST [23] which is an open source automatic test generation tool for C. Both the tools can use any SMT solver. To make a fair comparison we chose the same solver, Yices [19], for both. The experiments were performed on 3.2GHz Intel® Core™Duo machine with 1 GB RAM.

Table 5.1 shows the coverage obtained after Phase 1. The name of each test case reported in column 1 denotes the number of conditions present in that test case. For example test case tc50 has 50 conditions, i.e., 100 branches. We then report the number of iterations (# of SMT solver calls) performed in Phase 1, the % of branch coverage achieved, # of the branches covered, # of infeasible paths encountered and the time taken in Phase 1. The next 2 columns report the branches covered by the CFG and uniform_random strategies of

Table 5.1: Phase 1 Results

name	Phase 1					CREST-CFG	CREST-UR
	iterations	coverage (%)	branches covered	infeasible paths	time (s)	branches covered	branches covered
tc50	47	92	**92**	2	6.88	88	72
tc100	84	84	**168**	0	5.67	158	138
tc154	139	89	**276**	2	326.24	269	229
tc160	147	91	**293**	1	14.89	287	245
tc200	178	89	**356**	0	21.76	344	293
tc475	428	89	**851**	5	127.64	827	715
tc525	497	94	**988**	6	167.31	955	827
tc550	443	80	880	6	154.28	**919**	792
tc700	624	88	**1240**	10	292.35	1197	1059
tc750	451	59	892	9	215.49	**1141**	988
tc800	730	91	**1458**	2	406	1455	1217
tc875	686	77	1363	9	417.37	**1506**	1266
tc900	806	89	**1605**	7	521.79	1547	1339

CREST in the same number of iterations as that of our Phase 1. It can be clearly seen that in most of the cases the coverage achieved by Phase 1 is over 80% and this coverage is achieved in a very short amount of time. Also, the # of branches covered by Phase 1 is already greater than that of the strategies of CREST for the same number of iterations for most cases.

Table 5.2 shows the results after Phase 2 is applied. The first column reports the name of the test case. The next 4 columns report the # of iterations, # of branches covered, # of infeasible paths encountered, and the time taken by the basic DFS-based reachability guided strategy. The next 4 columns report the corresponding numbers for the 2-phase strategy. However, the numbers reported in all columns of our 2-phase approach are cumulative and include both Phase 1 and Phase 2. The next column shows the speed up of our 2-phase strategy over the DFS based reachability guided strategy. The last column reports the

Table 5.2: Phase 2 Results

name	DFS based reachability guided				2-Phase					CREST-CFG
	iterations	branches covered	inf. paths	time (s)	iterations	branches covered	inf. paths	time (s)	speed-up	iterations
tc50	73	94	9	10.69	51	94	2	7.71	**1.38**	51
tc100	84	168	0	7.13	84	168	0	5.77	**1.23**	90
tc154	144	288	0	399.52	146	288	2	348.92	**1.14**	142
tc160	151	300	151	24.88	152	300	1	16.54	**1.50**	151
tc200	61315	195	30651	-	178	356	0	22.21	$>$ **324.17**	182
tc475	22561	338	6981	-	445	867	6	187.71	$>$ **38.00**	480
tc525	17070	455	7545	-	508	997	6	211.69	$>$ **34.01**	527
tc550	8198	626	3993	-	**509**	990	6	279.5	$>$ **25.76**	3476
tc700	16169	450	5787	-	644	1253	13	468.96	$>$ **15.35**	659
tc750	17632	459	8850	-	**691**	1326	14	683.82	$>$ **10.52**	11460
tc800	14689	494	5899	-	741	1462	4	608.57	$>$ **11.83**	734
tc875	13759	589	6794	-	**802**	1570	11	845.2	$>$ **8.51**	12088
tc900	12987	572	5598	-	828	1624	9	762.43	$>$ **9.44**	828

'-': timeout of 2 hrs

number of iterations needed for achieving complete coverage by the CFG strategy of CREST. A time out of 2 hrs was set while performing the experiments.

By comparing the execution time and the number of iterations (i.e., # of SMT solver calls) needed by both the strategies, we found that for the same number of solver calls, the new strategy is often faster because of smaller path prefixes used in Phase 1. For example, consider test case tc160 in Table 5.2, the DFS based reachability-guided strategy took 24.88 sec for 151 solver calls whereas the new strategy took just 16.54 sec for 152 solver calls. Also, as discussed earlier, if no infeasible path is encountered in Phase 1, then complete coverage is achieved in Phase1. For example, consider test case tc100 in Table 5.1 and Table 5.2. The number of iterations reported in Phase 1 and at the end of Phase 2 for tc100 are same because as there are no infeasible paths in Phase 1, complete coverage is achieved in Phase 1 and there was no need for Phase 2.

It can also be seen that for all the test cases having 200 or more conditions, the DFS-based reachability guided strategy timed out. This is because of the huge number of infeasible paths encountered by the basic DFS strategy. In contrast, the 2-phase strategy encountered only a minimal number of infeasible paths and was able to achieve complete coverage. Also, the high coverages achieved in a very short time in Phase 1 significantly reduces the time required to achieve complete coverage. For tc200 and larger, significant speedup (generally $> 10\times$) has been achieved.

From the last column in Table 5.2, it can be seen that for some test cases our 2-phase strategy outperforms CREST. For example, for test cases tc750 and tc875 the number of iterations needed by CREST are 15 times more than our 2-phase strategy. This is mainly because of the intelligent conflict driven learning used in Phase 2. We do not compare the actual execution time between the two tools, because of the difference in the implementation of the two tools. However, the presented 2-phase strategy is not tool-specific and can be equally effective if implemented on any symbolic execution based test generation engines.

Chapter 6

Conclusion and Future Direction

Symbolic and Concolic execution based techniques are becoming popular for dynamic test generation. However, these techniques are not yet scalable to large programs. In this thesis, we proposed a 2-phase strategy for achieving high branch coverage quickly. We first identify edges and paths in the reachability graph that ought to be avoided in Phase 1, thus allowing us to avoid searching through longer paths that are deemed less promising. Phase 2 then explores the remaining branches using a DFS based approach with an intelligent conflict driven learning. This learning allows for a clever classification of unexplored branches with the help of unsat constraints. This helps in avoiding the exploration of many unnecessary paths.

The strategy was implemented on Intel's internal tool and was used to test large test programs that are used in high volume manufacturing of Intel's semiconductor devices. Experimental results show that Phase 1 alone achieves very high branch coverage (generally more than 80%) in a very short time. Sometimes Phase 1 alone is able to achieve a higher coverage than previous methods. Also, for the same number of solver calls, the 2-phase strategy takes less time than the DFS based strategy because of smaller path prefixes. Finally, the total number of SMT solver calls needed to achieve complete branch coverage is significantly lower for our strategy.

The strategy was also compared to the CFG directed strategy and uniform random search strategy of CREST. We observed that our strategy was able to achieve complete coverage in less number of iterations for most of the test cases.

Although the proposed flow is applicable to general classes of programs, the current conflict-driven-learning engine in Phase 2 targets stateless programs suitable for programs such as those used in ATEs. In the future, the conflict-driven learning technique can be generalized to broader classes of programs. In particular, instead of using only a control flow graph of the program, the data flow graph of the program can be used to check which variables change values along which paths. This information can then be used to check whether the unsat constraint found is applicable to all the sub paths between the conflicting branches or not. Further, instead of using BFS to mark edges in Phase 1, we can investigate the existence of better heuristics that will reduce the number of infeasible paths encountered in Phase 1, thus improving the coverage achieved at end of Phase 1.

Bibliography

[1] G. J. Myers, *Art of Software Testing.* New York, NY, USA: John Wiley & Sons, Inc., 1979.

[2] G. Tassy, "The economic impacts of inadequate infrastructure for software testing," tech. rep., National Institute of Standards and Technology, 2002.

[3] C. Pacheco and M. D. Ernst, "Eclat: Automatic Generation and Classification of Test Inputs," in *ECOOP*, pp. 504–527, 2005.

[4] B. Korel, "Automated Test Data Generation for Programs with Procedures," in *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software testing and analysis*, ISSTA '96, (New York, NY, USA), pp. 209–215, ACM, 1996.

[5] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An Automated Framework for Structural Test-Data Generation," in *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pp. 285 –288, Oct. 1998.

[6] J. Darringer and J. King, "Applications of Symbolic Execution to Program Testing," *Computer*, vol. 11, pp. 51 –60, April 1978.

[7] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT-A Formal System for Testing and Debugging Programs by Symbolic Execution," in *Proceedings of the International Conference on Reliable software*, (New York, NY, USA), pp. 234–245, ACM, 1975.

[8] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[9] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, (New York, NY, USA), pp. 263–272, ACM, 2005.

[10] P. D. Coward, "Symbolic Execution Systems - a review," *Softw. Eng. J.*, vol. 3, pp. 229–239, November 1988.

[11] G. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Compiler Construction* (R. Horspool, ed.), vol. 2304 of *Lecture Notes in Computer Science*, pp. 209–265, Springer Berlin / Heidelberg, 2002.

[12] I. Johnson, "Formal Verification with SMT solvers: Why and How."

[13] A. Cimatti, "Beyond Boolean SAT: Satisfiability Modulo Theories," in *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pp. 68 –73, May 2008.

[14] L. M. de Moura and N. Bjørner, "Satisfiability Modulo Theories: An Appetizer," in *SBMF*, pp. 23–36, 2009.

[15] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Dpll(t): Fast decision procedures," pp. 175–188, Springer, 2004.

[16] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *J. ACM*, vol. 7, pp. 201–215, July 1960.

[17] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem-Proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.

[18] B. Dutertre and L. M. de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," in *CAV*, pp. 81–94, 2006.

[19] B. Dutertre and L. D. Moura, "The Yices SMT solver," tech. rep., 2006.

[20] A. Griggio, "A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic," *JSAT*, vol. 8, pp. 1–27, January 2012.

[21] C. Barrett and C. Tinelli, "CVC3," in *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)* (W. Damm and H. Hermanns, eds.), vol. 4590 of *Lecture Notes in Computer Science*, pp. 298–302, Springer-Verlag, July 2007. Berlin, Germany.

[22] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, (New York, NY, USA), pp. 213–223, ACM, 2005.

[23] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pp. 443 –446, 2008.

[24] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test Input Generation with Java PathFinder," *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 97–107, July 2004.

[25] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, (New York, NY, USA), pp. 322–335, ACM, 2006.

[26] R. Majumdar and K. Sen, "Hybrid Concolic Testing," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pp. 416 –426, May 2007.

[27] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan, "Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs," *Asian Test Symposium*, pp. 59–64, 2010.

[28] S. Prabhu, M. S. Hsiao, S. Krishnamoorthy, L. Lingappan, V. Gangaram, and J. Grundy, "An Efficient 2-Phase Strategy to Achieve High Branch Coverage," in *20th Asian Test Symposium (ATS)*, pp. 167 –174, Nov. 2011.

[29] I. Lynce, J. Marques-Silva, and I. E. Desenvolvimento, "On Computing Minimum Unsatisfiable Cores," 2003.

[30] A. Cimatti, A. Griggio, and R. Sebastiani, "A Simple and Flexible way of Computing Small Unsatisfiable Cores in SAT Modulo Theories," in *Proceedings of the 10th international conference on Theory and applications of satisfiability testing*, SAT'07, pp. 334–339, Springer-Verlag, 2007.

[31] N. Dershowitz, Z. Hanna, and E. Nadel, "A Scalable Algorithm for Minimal Unsatisfiable Core Extraction," in *In Proc. SAT06*, Springer, 2006.

[32] M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. M. Silva, and K. A. Sakallah, "A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas," in *SAT*, pp. 467–474, 2005.

[33] Y. Oh, M. Mneimneh, Z. Andraus, K. Sakallah, and I. Markov, "AMUSE: A Minimally-Unsatisfiable Subformula Extractor," in *Design Automation Conference, 2004. Proceedings. 41st*, pp. 518 –523, July 2004.

[34] L. Zhang and S. Malik, "Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formula," in *SAT*, vol. 3, 2003.