Strategies for Scalable Symbolic Execution-based Test Generation

Saparya Krishnamoorthy

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
Patrick Schaumont
Paul Plassmann

June 28, 2010
Blacksburg, Virginia

Keywords: *software verification, dynamic test generation, symbolic execution, satisfiability modulo theories, path explosion*

Strategies for Scalable Symbolic Execution-based Test Generation

Saparya Krishnamoorthy

ABSTRACT

With the advent of advanced program analysis and constraint solving techniques, several test generation tools use variants of symbolic execution. Symbolic techniques have been shown to be very effective in path-based test generation; however, they fail to scale to large programs due to the exponential number of paths to be explored. In this thesis, we focus on tackling this path explosion problem and propose search strategies to achieve quick branch coverage under symbolic execution, while exploring only a fraction of paths in the program. We present a reachability-guided strategy that makes use of the reachability graph of the program to explore unvisited portions of the program and a conflict-driven backtracking strategy that utilizes conflict analysis to perform nonchronological backtracking. We also propose error-directed search strategies, that are aimed at catching bugs in the program faster, by targeting those parts of the program where bugs are likely to be found or those that are hard to reach. We present experimental evidence that these strategies can significantly reduce the search space and improve the speed of test generation for programs.

To my family

Acknowledgments

Any record of work is incomplete without an expression of gratitude towards those who made it possible. At the outset, I owe my deepest gratitude to my advisor, Prof. Michael S. Hsiao. I am grateful to him for his academic guidance, continued encouragement and above all, for being a constant source of inspiration.

I thank Prof. Patrick Schaumont and Prof. Paul Plassmann for graciously agreeing to serve on my thesis committee. I also thank the administrative staff of the Graduate School and the Bradley Department of Electrical and Computer Engineering for their timely help and continued cooperation with paperwork and other administrative matters.

I would like to express my gratitude to Vijay Gangaram and Loganathan Lingappan for giving me an opportunity to intern in their team at Intel Corporation and to work on challenging problems towards my thesis. I am also grateful to them for supporting this work with a research grant. I thank Jim D. Grundy for taking the time to explain to me, several concepts of symbolic execution-driven test generation. I learned a lot from my interactions with them, during the seven months of my internship at Intel Corporation.

I am thankful to my friends in the PROACTIVE Research Group for enriching my research experience - Maheshwar Chandrasekhar, Min Li, Neha Goel, Harini Jagadeesan, Nikhil Rahagude, Sarvesh Prabhu, Dhrumeel Bakshi and Supratik Misra.

My heartfelt thanks to my friends, Mrudula P. Karve, Kavita H. Poddar, Vaishnavi Srinivasaraghavan, Sushrutha Vigraham, Shruti V. Iyer and Aditi Chaudhry for making my

two years at Virginia Tech unforgettable. I will always cherish the moments that we spent together.

Finally, I wish to thank my parents K.R. Krishnamoorthy and Bharati Krishnamoorthy and my sister, Pavitra for their unconditional love, support and encouragement. Nothing would have been possible without my family, because of whom, I am what I am today.

Saparya Krishnamoorthy

Blacksburg

June 23, 2010.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Today, testing is widely recognized as a crucial step in software development as the primary method to verify the correct functioning of software programs. The correct functioning of software is in turn of utmost importance, especially in safety-critical applications. Efficient testing of software programs is inherently a difficult process, due to the complexity of software. Consequently, testing usually accounts for about 50% of the software development cost [1]. According to a Planning Report made by the National Institute of Standards and Technology [2], software failures currently cost the US economy alone about $60 billion every year, and improvements in software testing infrastructure might save one-third of the cost.

Efficient testing of a program requires the application of relevant test inputs to the program-under-test. Among the various kinds of testing usually performed during the software development cycle, unit testing applies to the individual components of a software system. A program is decomposed into units, where each unit is a set of functions, and these units are independently tested. This kind of testing requires the application of test inputs to the unit. Manual testing is labor-intensive and cannot guarantee that all possible behaviors of the program have been verified [3]. To improve the observed test coverage, several techniques have been proposed to automatically generate values for the inputs. One such technique is to randomly choose the values over the domain of potential inputs [4]. The problem

with this technique is that many sets of values may result in the same observable output and are thus redundant, and the probability of choosing inputs that cause faulty behaviour may be extremely small. Recently, many approaches have been proposed to address the problem of redundant input values and to increase test coverage: search-based testing [5] and constraint-based testing [6]. The former approach is based on exploring the input space of the program using optimization-like methods to guide the search towards relevant test data. Constraint-based testing [6] is an increasingly popular technique to automatically generate test input values and tackle some of the previously mentioned challenges. This method focuses on translating portions of the program into logical formulas, whose solutions are relevant test data.

Constraint-based testing is powered by recent progress in constraint solvers and symbolic execution engines [7]. Such symbolic techniques have been shown to be very effective in path-based test generation; however, they fail to scale to large programs [8]. This is because the possible number of execution paths to be considered symbolically is so large that, only a small part of the program path space is actually explored. This phenomenon in path-based testing is called the path explosion phenomenon [9].

Covering all paths of a program is however, not the primary objective of most modern testing practices. As an example, we cite the tool Bullseye which is widely used [10] in the software industry to measure the quality of test data. This tool reports code coverage as only function and branch coverage [11]. Keeping these points in mind, we propose strategies to quickly achieve complete branch coverage of the program-under-test using symbolic engine-driven test generation engines.

1.1 Contributions of this Thesis

In this thesis, we focus on tackling the path explosion problem in symbolic execution-based testing, and aim to reduce the time for test generation. We propose symbolic search strategies

that help achieve branch coverage quickly, while searching only a small fraction of paths in the program. We present search strategies as enhancements of the basic depth-first search (DFS) procedure: a reachability-guided strategy that utilizes the program reachability graph to explore unvisited parts of the program and a conflict-driven backtracking strategy that makes use of conflict analysis to enable backtracking over multiple levels of the graph in order to quickly discard as many redundant paths as possible. We also propose error-directed search strategies, that are aimed at catching bugs in the program faster, by targeting those parts of the program where bugs are likely to be found or that are hard to reach. Towards this goal, we propose an assertion-directed search strategy and a *detectability*-based heuristic aimed at targeting hard-to-detect errors in the program-under-test.

Our techniques have been implemented in a path-based testing framework and experiments have been carried out to evaluate their efficiency. Experiments show that our search strategies are effective in discarding infeasible paths and reducing the number of paths explored, without a loss of branch coverage compared to the base search strategy.

The research contributions of this thesis can be summarized as follows:

1. A reachability-guided search strategy to address the path explosion phenomenon in symbolic execution-based testing. This strategy uses static information about the program-under-test from its control flow graph (CFG) to achieve branch coverage of the program quickly.

2. A conflict-driven backtracking strategy that uses conflict analysis to determine the cause of an encountered path being declared infeasible and performs nonchronological backtracking to the cause of the infeasibility.

3. Error-directed strategies aimed at catching errors quickly in the program-under-test, by targeting those parts of the program where bugs are likely to be found or that are hard to reach.

1.2 Thesis Outline

The remainder of this thesis is structured as follows.

Chapter 2: This chapter introduces the concepts of symbolic execution-based test generation using SMT solvers. This chapter also describes the path explosion phenomenon in path-based testing and surveys the various techniques proposed in literature to address this scalability challenge.

Chapter 3: This chapter describes our proposed reachability-guided search strategy.

Chapter 4: Our proposed conflict-driven backtracking strategy is detailed in this chapter. The process of nonchronological backtracking and conflict-driven learning are explained in detail.

Chapter 5: This chapter explains our strategies aimed at targeting program errors quickly. The assertion-directed strategy and the *detectability*-based heuristic directed towards hard-to-detect errors are described here.

Chapter 6: Our implementation and experimental results are described in this chapter.

Chapter 7: This chapter concludes this thesis, along with suggestions for future work.

Chapter 2

Background

This chapter is devoted to explaining the concepts that are required to understand the work done towards this thesis. This chapter also describes the previous work done in this area and how it was used towards this thesis. The following topics are described:

1. Automated Testing
2. Symbolic Execution Systems
3. Using SMT solvers for Constraint Solving
4. Dynamic Test Generation Engines
5. Path Explosion Phenomenon
6. Related Work

2.1 Automated Testing

Testing, today, is the primary way to check the correctness of software. Testing usually accounts for about 50% of the cost of software development [1]. According to a Planning

Report made by the National Institute of Standards and Technology [2], software failures currently cost the US economy alone about $60 billion every year, and improvements in software testing infrastructure might save one-third of the cost.

Among the various kinds of testing usually performed during the software development cycle, unit testing applies to the individual components of a software system. A program is decomposed into units, where each unit is a set of functions, and these units are independently tested. This kind of testing requires the application of test inputs to the unit. Manually specifying these values is labor-intensive and it cannot guarantee that all possible behaviors of the unit will be observed during testing [3].

In order to improve the test coverage observed, several techniques have been proposed to automatically generate values for the inputs. One such technique is to randomly choose the values over the domain of potential inputs [4]. The problem with this technique is that many sets of values may lead to the same observable behavior and are thus redundant, and the probability of selecting particular inputs that cause faulty behaviour may be extremely small.

There are some approaches that address the problem of redundant input values and increase test coverage: search-based testing [5] and constraint-based testing [6]. The former approach is based on exploring the input space of the program using optimization-like methods to guide the search towards relevant test data, whereas the latter focuses on translating parts of the program into logical formulae whose solutions are relevant test data [12]. Here, we focus on constraint-based testing, which is powered by recent progress in constraint solvers and symbolic execution engines. These concepts are described in detail in the following sections.

2.2 Symbolic Execution Systems

The past two decades have seen an increasing interest in formal methods of software development and verification. Symbolic execution is a major component of these developments as it can be used to aid software testing and program proving. Symbolic execution, sometimes

called symbolic evaluation, does not execute the program in the traditional sense of the word. As described in [7], the notion of execution requires that a selection of paths through the program are exercised by a set of data values. A program that is executed using actual data results in the output of a series of values, whereas in symbolic execution the data is replaced by symbolic values, and a set of expressions, one per output variable is produced.

The most common approach to symbolic execution is to perform an analysis of the program, resulting in the creation of a flow graph. A flow graph is a representation of a program which identifies the decision points and assignments associated with each branch. By traversing this flow graph from an entry point, along a particular path, a list of assignment statements and branch predicates is produced. The resulting path is represented by a series of input variables, condition predicates and assignment statements. Symbolic execution involves following this path from beginning to end.

During this path traversal each input variable is given a symbol in place of the actual value. Thereafter, each assignment statement is evaluated so that it is expressed in terms of symbolic values of input variables and constraints. At the end of symbolic execution of a path, the output variables will be represented by expressions in terms of symbolic values of input variables and constraints. A list of these constraints, known as path conditions, is provided by the set of symbolic representations of each conditional predicate along the path. Analysis of these constraints may indicate that the path is not executable owing to a contradiction.

In symbolic execution, a program is executed on symbolic inputs: the execution of an assignment statement updates the program state with symbolic expressions and the execution of a conditional expression generates a symbolic constraint in terms of the symbolic inputs. Values are then generated that satisfy the symbolic constraints generated along each execution path. Such an input forces the program to take that execution path during normal testing [8]. The substituted values constitute the test case and the evaluation of the expression provides the corresponding output value. In other words, path-oriented approaches to testing are built upon the idea of a path predicate. In [12], a path predicate is defined as the following:

Definition 1 (Path Predicate). *Given a program P of input domain D and π a path of P, a path predicate of π is a formula φ_π on D such that if $V \models \varphi_\pi$ then execution of P on V follows the path π.*

A path predicate for φ_π for a path π can be computed by keeping track of logical relations among variables along the execution. A solution to a path predicate φ_π for a given program P is actually a test case exercising path π. The process of symbolic execution with the help of path predicates can be best explained with an example. Table 2.1 shows how constraint-based execution is performed along a path of a program corresponding to the C program illustrated in Figure 2.2. Note that the branch predicates are represented in single static assignment (SSA) form. The branch predicates are computed in terms of the symbolic variables defined in place of the actual program variables. The path predicate of the shown path $0 \rightarrow 1 \rightarrow 2 \rightarrow (3, true) \rightarrow (4, false)$ is computed with the help of the branch constraints of the path as: $\langle y_1 = y_0 + 1\rangle \wedge \langle z_0 = 3x_0\rangle \wedge \langle x_0 \neq y_1\rangle \wedge \langle z_0 \neq x_0 + 6\rangle$. This, in turn, can be represented in terms of the input variables as: $\langle x_0 \neq y_0 + 1\rangle \wedge \langle 3x_0 \neq x_0 + 6\rangle$.

Table 2.1: *Constraint-based execution along a path in a program*

Line	Instruction	Branch constraint
0	input (x, y)	new vars x_0, y_0
1	y := y + 1	$y_1 = y_0 + 1$
2	z := 3 * x	$z_0 = 3x_0$
3	if (x != y) - *true*	$x_0 \neq y_1$
4	if (z == x + 6) - *false*	$z_0 \neq x_0 + 6$

This path constraint represents all the input vectors that drive the program through the path shown in Table 2.1. To force the program through a different path by taking a different branch on this path, the instrumented program calculates a solution to the path constraint obtained by negating a branch predicate of the current path constraint. By repeating this process, the test generation tool attempts to sweep through all the paths of the program [6]. The interested reader is referred to [7] for more details on symbolic execution systems.

Recently, concolic testing [3, 6] have been proposed as a variant of symbolic execution where symbolic execution is performed simultaneously with concrete execution. Specifically, the

program is simultaneously executed on concrete and symbolic values, and symbolic constraints generated along the path are simplified using the corresponding concrete values.

Thus, several symbolic techniques have been proposed for test data generation. This method of automated test generation addresses the limitations of manual and random testing. Symbolic techniques provide better coverage of a program's behavior because they try to generate a single test input for each feasible execution path. The next section explains how the path constraints that arise in symbolic execution are solved.

2.3 Using SMT Solvers for Constraint Solving

As described earlier, test data to exercise a particular path in a program are generated by solving the path constraints of that path. Constraint solvers, such as Satisfiability Modulo Theories (SMT) solvers, can be used to solve these path constraints. Such a solver should be able to check if a constraint equation is 'satisfiable' or 'unsatisfiable'. In the case of satisfiability, it should be capable of returning a solution to the constraint equation. In this section, we describe SMT solvers and how they can be used to solve the path constraints that arise in symbolic execution.

The Satisfiability Modulo Theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. Examples of theories typically used in computer science 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. Formally speaking, an SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations, and SMT is the problem of determining whether such a formula is satisfiable. In other words, the SMT problem can be interpreted as an instance of the Boolean satisfiability problem (SAT) in which some of the binary variables are replaced by predicates over a suitable set of non-binary variables.

The current SMT technology is built upon 40 years of theory and empirical studies. Most of the solvers today are based on the Davis-Putnam procedure that was formulated in 1960 [13], which was then revised for performance in 1962 [14]. This procedure is thus known as the DPLL (Davis-Putnam-Logemann-Loveland) algorithm. These satisfiability solvers utilize the separation of clauses in the conjunctive normal form (CNF) of propositional formulas. In Boolean logic, a formula is in CNF if it is a conjunction of clauses, where a clause is a disjunction of literals, and a literal is in turn, an atomic formula (atom) or its negation. A CNF formula is similar to the canonical product of sums form used in circuit theory. The CNF formula is said to be satisiable if there exists an assignment of truth values to the used Boolean variables in order for the entire formula to be true. If there is no such assignment, the formula is deemed unsatisfiable.

The SMT framework changes from the propositional case covered by the DPLL procedure by changing the propositional atoms to first order sentences, i.e., first order logic formulas with no free variables. This approach relies on a modern SAT procedure to deal with the boolean search, and on dedicated theory solvers which are able to decide the satisfiability of theory atoms with respect to the background theory [15, 16, 17, 18]. In other words, SMT solvers decide satisfiability of functions modulo theories. A theory T is a set of rules of inference in which first order logic predicates are interpreted. 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. In this way, SMT solvers are based on the tight integration of propositional SAT solvers with procedures to reason about the theory component [19].

Although satisfiability is the classic NP-complete problem, and modulo theories possibly even more complex, the gap between engineered programs and theoretically possible programs is rather large, and the technology that currently exists for SMT is overwhelmingly fast. A large variety of real world applications can be modeled with just linear transformations, so the theories that reason about linear arithmetic are still powerful today. Also, there already exist several application domains in the propositional sense of satisfiability, so the added layer of abstraction with SMT can improve performance significantly [20].

Satisfiability-modulo theory (SMT) solvers are now widely being used as constraint-solvers for program test generation as well as other formal verification techniques. This is because, many programs can be automatically translated from code to formulas that an SMT solver can decide. Various areas of first order logic and arithmetic are decidable and we can reason about programs in such theories. Also, the theory to be used can be chosen based on the complexity of the problem in question. Linear arithmetic, uninterpreted functions and arrays are examples of some useful decidable theories that we can apply to test a program [20]. In our context of symbolic execution of a program, the path constraint of a path, say P, of the program can be expressed in terms of these decidable theories. This path constraint is then solved by an SMT solver and the solution thus generated (if such a solution exists) acts as the test data to exercise the path P in the program.

2.4 Dynamic Test Generation Engines

With the advent of advanced program analysis and constraint solving techniques, several tools today, like DART [6], CUTE [3], etc., utilize symbolic reasoning for dynamic test generation. These tools address the limitation of unit-testing, namely the need to write test driver and harness code to simulate the external environment of a software application. They are able to gather knowledge about the execution of a program using a directed search. Starting with a random input (sometimes called a *seed*),[1] these tools help calculate during each execution an input vector for the next execution. This vector contains values that are the solution of the symbolic constraints gathered from predicates in branch statements during the previous execution. This new vector tries to force execution of the program through a new path. By repeating this process, the tool attempts to force the program to sweep through all its feasible paths [6]. The complete process of dynamic test generation using symbolic execution, can be seen in Figure 2.1.

[1]Such a *seed* however, is not required when we have control-flow information about the program-under-test.

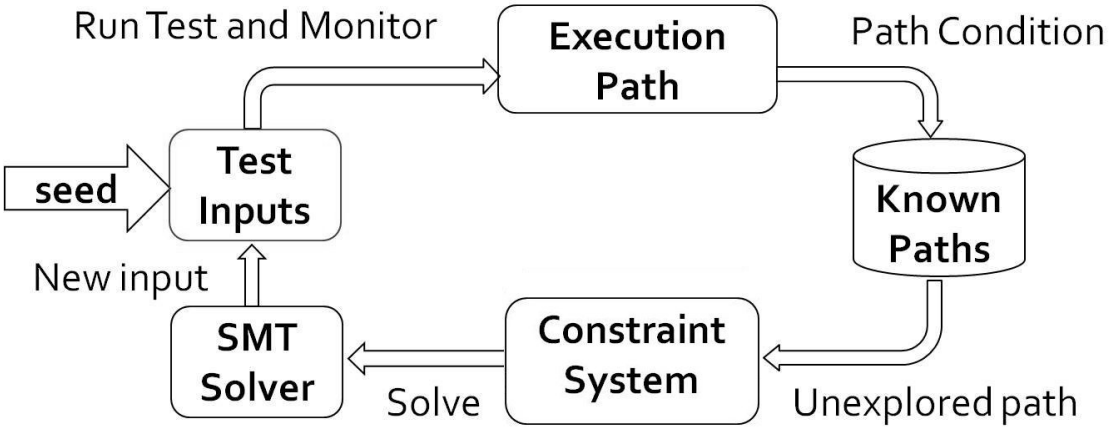

Figure 2.1: *The process of test generation using symbolic execution*

2.4.1 Program Instrumentation

To test a program, the tool first instruments the program-under-test. Program instrumentation provides a way to observe program execution. The test program is modified so that it emits trace information as it runs. This can be achieved with the help of tools like CIL (C Intermediate Language) [21], an OCaml application for parsing and analyzing C code. Then the instrumented program is repeatedly run, with different test inputs for each execution.

The C programming language is well-known for its flexibility in dealing with low-level constructs. Unfortunately, it is also equally well-known for being difficult to understand and analyze, both by humans and by automated tools. Thus, in order to efficiently observe and analyze the execution of the program-under-test, we require a good intermediate language. It should be simple to analyze, close to the source and able to handle real-world code. CIL (C Intermediate Language) is a highly-structured clean subset of C that meets these requirements.

CIL features a reduced number of syntactic and conceptual forms; for example, all looping constructs are reduced to a single form and all function bodies are given explicit return statements. CIL removes from the source file all those type declarations, local variables and

inline functions that are not used in the file. In other words, our analysis does not have to see all the information from the header files. CIL also separates type declarations from code, makes type promotions explicit and flattens scopes within function bodies. Shortcut evaluation of boolean expressions and the ?: operator are compiled into explicit conditionals. In essence, CIL compiles all valid C programs into a few core constructs with very clean semantics. This reduces the number of cases that must be considered when manipulating a C program, making it more amenable to analysis and transformation. Many of these steps are carried out at some stage by most C compilers, but CIL makes analysis easier by exposing more structure in the abstract syntax.

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 (intraprocedural) control flow. Statements capture local control flow. CIL provides both high-level program structure and low-level control-flow information. The program structure is captured by a recursive structure of statements, with every statement annotated with successor and predecessor control-flow information. This simple program representation can be used with routines that require an abstract syntax tree or AST (e.g., type-based analyses or pretty-printers), as well as with routines that require a control flow graph or CFG (e.g., dataflow analyses).

The CIL manual describes some of the transformations that are applied to C programs to convert them to CIL [22]. We have reproduced some of them here to show how the analysis becomes simpler, since there are fewer and simpler C constructs to be dealt with.

Example 1. One of the most significant transformations is that expressions that contain side-effects are separated into statements.

```
int main(void) {
{
    int x, f(int);
    return (x ++ + f(x));
}
```

CIL Output:

```
/* Generated by CIL v. 1.3.7 */
#line 1 "cilcode.tmp/ex1.c"
extern int f(int  ) ;
#line 1 "cilcode.tmp/ex1.c"
int main(void)
{
  int x ;
  int tmp ;
  int tmp___0 ;
  {
#line 2
    tmp = x;
#line 2
    x ++;
#line 2
    tmp___0 = f(x);
#line 2
    return (tmp + tmp___0);
  }
}
```

Internally, the x ++ statement is turned into an assignment which is printed like the original. CIL has only three forms of basic statements: assignments, function calls and inline assembly.

CIL also stores location information with all statements and can take advantage of this information by inserting #line directives in the resulting output. This allows events in a heavily-transformed program to be easily lined up with the correct source line in the original program.

Example 2. All forms of loops (while, for and do) are compiled internally as a single **while(1)** looping construct with explicit break statements for termination. For simple *while* loops, the pretty printer is able to print back the original loop, even though it is internally compiled as **while(1)**.

```
int main(void)
{
    int x, y;
    for(int i = 0; i<5; i++) {
        if(i == 5) continue;
        if(i == 4) break;
        i += 2;
    }
    while(x < 5) {
        if(x == 3) continue;
        x ++;
    }
}
```

CIL Output:

```
/* Generated by CIL v. 1.3.7 */
#line 1 "cilcode.tmp/ex2.c"
int main(void)
{
  int x ;
  int i ;
  {
#line 2
    i = 0;
#line 2
    while (i < 5) {
#line 3
        if (i == 5) {
            goto __Cont;
        }
#line 4
        if (i == 4) {
#line 4
            break;
        }
#line 5
```

```
        i += 2;
         __Cont: /* CIL Label */
#line 2
        i ++;
    }
#line 7
    while (x < 5) {
#line 8
        if (x == 3) {
#line 8
            continue;
        }
#line 9
        x ++;
    }
#line 11
    return (0);
  }
}
```

CIL's conceptual design tries to stay close to C, so that conclusions about a CIL program can be mapped back to statements about the source program. Additionally, translating from CIL to C is fairly easy, including reconstruction of common C syntactic idioms. Finally, a key requirement for our application is the ability to parse and represent the variety of constructs which occur in real-world systems code, such as compiler-specific extensions and inline assembly. CIL supports all GCC and MSVC extensions except for nested functions, and it can handle the entire Linux kernel.

Given a program, the external interfaces through which the program can obtain inputs via uninitialized memory locations are identified. For each external interface, we determine the type of the input that can be passed to the program via that interface. In C, a type is defined recursively as either a basic type (int, float, char, enum, etc.), a struct type composed of one or more fields of other types, an array of some type, or a pointer to some type. These external interfaces of a C program can be easily determined and instrumented by a light-weight static

parsing of the program's source code [21].

As a part of the program instrumentation process for symbolic execution, after the code is transformed with the help of tools like CIL into a simplified form that can be easily analyzed, the concrete variables and the operations on them have to be changed to their symbolic counterparts. Usually, the test generation tool will provide classes for manipulating different symbolic expressions (integers, strings, arrays etc.). These classes would contain the methods that operate on symbolic variables (e.g. different comparison operators, addition and multiplication for integers). Statements accessing or updating variables also need to be instrumented. This is done with the help of tags that describe that the variable has been initialized [23].

During instrumentation, function calls are inserted into the C program-under-test, which (loosely) correspond to an execution of the program by a stack machine. These calls are then used to symbolically execute the program, by maintaining a symbolic stack along with a symbolic memory map. A C expression (with no side effects) generates a series of Load and Store calls corresponding to the 'postfix' evaluation of the expression, using a stack (i.e., a Load indicates that a value is pushed onto the stack, and unary and binary operations are applied to one/two values popped off the stack). Entering the *then* or *else* block of an *if* statement generates a Branch call indicating which branch was taken. An assignment statement generates a single Store call, indicating that a value is popped off the stack and stored in the given address [8].

In order to symbolically execute the C program-under-test by a stack machine, the inputs to the program are defined as memory locations which are dynamically initialized at runtime through the static external interface. The memory M is a mapping from memory addresses m to, say, 32-bit words. The symbolic variables are defined by their addresses. Thus, in an expression, m denotes either a memory address or the symbolic variable identified by address m, depending on the context. The program P manipulates the memory through statements that are specially tailored abstractions of the machine instructions actually executed. A

statement can be a conditional statement c of the form *if(e) then goto l'* (where e is an expression over symbolic variables and *l'* is a statement label), an assignment statement a of the form $m \leftarrow e$ (where m is a memory address), **abort**, corresponding to a program error or **halt**, corresponding to normal termination.

A program P defines a sequence of input addresses $\vec{M_0}$, the addresses of the input parameters of P. An input vector $\vec{I}$, which associates a value to each input parameter, defines the initial value of $\vec{M_0}$. The semantics of P at the RAM machine level allows us to define for each input vector $\vec{I}$ an execution sequence: the result of executing P on $\vec{I}$. Let $Execs(P)$ be the set of such executions generated by all possible $\vec{I}$. By viewing each statement as a node, $Execs(P)$ forms a tree, called the execution tree. Its assignment nodes have one successor; its conditional nodes have one or two successors; and its leaves are labeled **abort** or **halt**.

Once code instrumentation is completed, the test generation tool performs symbolic execution of the instrumented program. During execution, the test generation tool monitors the execution trace and records information about the current path being executed (like coverage etc.). The branches along the path being executed are identified with the help of the identifiers added to the code by CIL or a similar transformation tool.

2.4.2 Program Path Exploration using the Basic DFS Procedure

The goal of a dynamic test generation engine is to explore all paths in the execution tree of the program. The program execution tree represents the unfolded control-flow graph of the program. Each node in this graph corresponds to a conditional or a choice point in the program (*if*, *switch*, etc.), and each of these nodes has two possible edges corresponding to its *true* and *false* branches. To carry out a search through the execution tree, the instrumented program is run repeatedly. A new path is chosen for each run, its path predicate is solved and the obtained solution (if any) is stored as a test vector. This procedure is repeated until all paths have been explored. The test input values are obtained with the help of a constraint solver. If the solver is able to find a feasible solution for the instance, it provides the test

generation tool with the solution that it found to be used as the test input in the next run. Otherwise the solver returns 'unsatisfiable', making the new path infeasible to explore [6].

Algorithm 2.1 *DFS-based path exploration strategy*

```
Path ← ∅, Tests ← ∅
CurrNode ← initial node
add CurrNode to Path
while Path is not empty do
  if not covered any branches of CurrNode then
    transition ← false branch of CurrNode
    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
  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
```

The most common method to explore the program execution tree is the depth-first search (DFS) strategy. Algorithm 2.1 shows the steps involved in the basic DFS strategy. In this algorithm, we assume that we have the static CFG of the program-under-test. In this scenario, we do not require a random *seed* to initiate the path exploration. Here, we select the *false* branch at every control point in the program, making the *all-false* path always the first explored path. The path predicate of each path is built incrementally, reusing the path prefix up to the last choice point in the program. When the program halts at the end of a run, new input values are generated to attempt to force the next run to explore the last unexplored branch of the conditional on the stack. For conditionals, we force the search to take the *if* or *else* branch by adding to the current path predicate ϕ, the condition predicate *cond* or its negation $\neg cond$.

2.4.3 A Dynamic Test Generation Example

```
int g(int x, int y)
{
    y++;
    z = 3 * x;
    if (x != y) {
        if (z == x + 6)
            abort();      // Error
    }
    return 0;
}
```

Figure 2.2: *An example C program to demonstrate symbolic execution*

We describe the basic depth-first search strategy in the context of the C program shown in Figure 2.2. The function 'g' is defective because it may lead to an **abort** statement for some value of its input vector. Here, the **abort** statement represents a failure state that occurs in the event of an error in the program. Let us assume that a symbolic test generation engine initially generates the value 0 for both x and y. As a result, 'g' executes the *false* branch of the first if-statement. The path predicate $\langle x_0 \neq y_0 \rangle$ is formed on the fly, based on the evaluation of the path conditions. To force the program through a different path, the instrumented 'g' calculates a solution to the path constraint $\langle x_0 = y_0 \rangle$, obtained by negating the current path constraint. A possible solution to this path constraint is $(x_0 = 75, y_0 = 0)$ and consequently the *true* branch of the first if-statement and the *false* branch of the second if-statement are executed. The predicate sequence $\langle x_0 \neq y_0 + 1 \rangle \wedge \langle 3x_0 \neq x_0 + 6 \rangle$ is the path constraint. Following the DFS strategy, the last predicate of the current path constraint is negated, which results in $\langle x_0 \neq y_0 + 1 \rangle \wedge \langle 3x_0 = x_0 + 6 \rangle$. A possible solution to this new path constraint is $(x_0 = 3, y_0 = 86)$; say, the instrumented 'g' uses these values for the input variables, when it runs again. This execution reveals the error in the program by driving it into the **abort()** statement. In this way, bugs are uncovered as the paths of the program are explored in a depth-first manner.

2.5 Path Explosion Phenomenon

Symbolic techniques have been shown to be very effective in path-based test case generation; however, they fail to scale to large programs [8]. This is because the possible number of execution paths to be considered symbolically is so large that finally, only a small part of the program path space is actually explored. Thus, it is important to devise search strategies that help achieve branch coverage quickly despite searching only a small fraction of paths in the program.

The most significant scalability challenge faced by path-based testing is how to handle these exponential number of paths in the program. The path explosion happens mainly because of nested calls, loops and conditions. Moreover, path-based methods often require a bound k on the length of paths to explore, and the number of paths may increase considerably if k is overestimated. This is all the more problematic in programs where some parts of the program can be reached only with very long paths, which is a common situation [12].

The following example illustrates the path explosion problem. We consider a program that checks whether an input string of a particular length matches a particular pattern. The function 'pattern-match(string, pattern)' checks if the current string matches the pattern. For simplicity, we restrict our strings to Boolean strings, i.e., each element of the string can be either 0 or 1. The *pattern-match* function checks if the supplied string of a given length matches the pattern, which is specified in terms of a regular expression (e.g. $/1+0+/$). We use the dynamic test generation tool to generate tests for the program, i.e., different strings of the given length.

We notice that, as the length of the string is increased, the run-time of the test-generation process increases rapidly. This is because the tool tries all possible values for *every* element of the string, in order to explore all the paths of the program. In other words, for a string of length n, there are 2^n possible combinations (or paths in the program), and consequently those many times that the SMT solver is 'called' by the test generation tool. These results

are depicted in Figure 2.3.

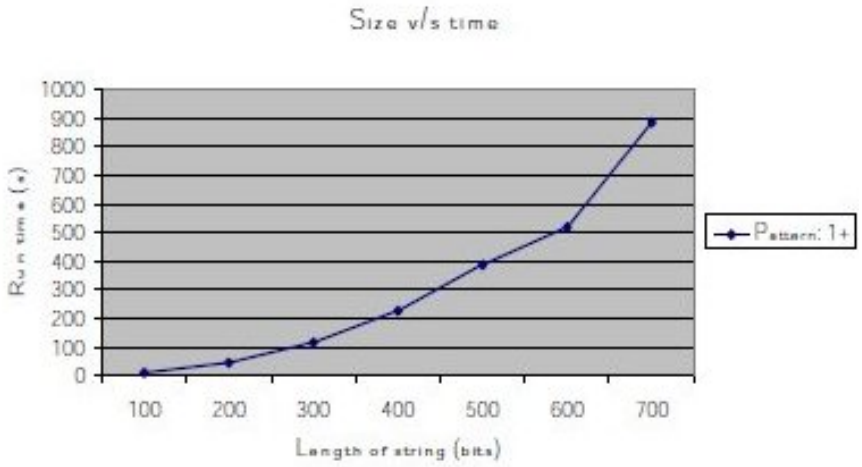

Figure 2.3: *Comparison of the execution time of the pattern-matching program with respect to the length of the string*

We also notice that as the complexity of the pattern to be matched increases, the run-time again increases. This is because the test generation tool explores all the paths within the *pattern-match* function. This can be seen graphically in Figure 2.4.

However, covering all paths of a program is not the primary objective of current testing practices. Even in critical systems, it is only required to fully cover a class of structural entities of the program source code such as instructions, branches or conditionals. As an example, for testing the afore-mentioned pattern-matching function, we might only be interested in a test case that matches a given pattern and another that does not. We may not be interested in the rest of the exponential number of possible combinations.

In this thesis, we focus on tackling the path explosion problem in path-based testing. The aim is to provide heuristics to discard as many irrelevant paths as possible, thus reducing the number of solver calls and the whole computation time. We study and investigate different methods to overcome the path explosion problem. We aim to reduce the compute time spent in exploring paths within helper functions and improve the speed of test generation. We do this by employing smarter search strategies, in place of depth-first search.

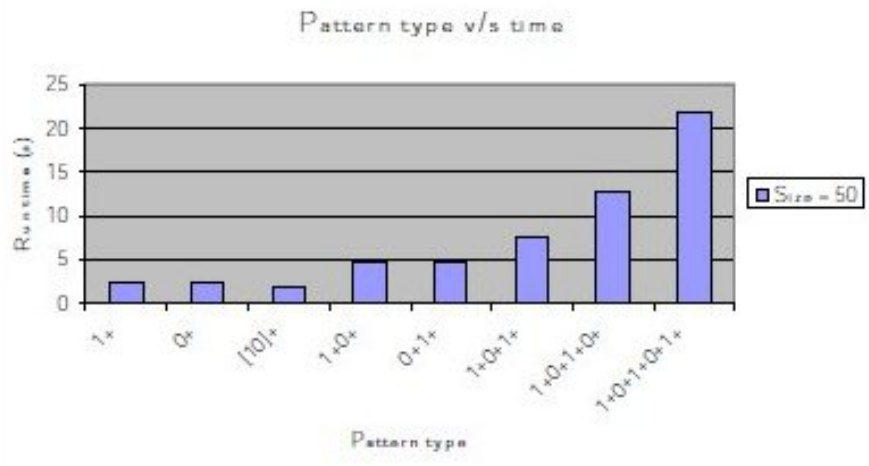

Figure 2.4: *Comparison of the execution time of the pattern-matching program with respect to the complexity of the pattern*

2.6 Related Work

In the last five years, several tools and techniques have been proposed that automatically generate test inputs. Consequently, many other lines of work address the problem of path explosion. A technique described in [8] builds on the basic depth-first search strategy to bound the number of branches explored. Burnim and Sen [8] also describe other strategies like the uniform random search and control-flow graph (CFG) directed search. In this CFG-directed search, the algorithm uses the static grapth of the program to find short paths from branches along the execution to branches that have not yet been explored.

In [24], a technique called *hybrid testing* was proposed, which deals with breaking the regularity of a DFS with random test generation. During the DFS process, a test datum is sometimes generated at random and the DFS continues from the corresponding path. *Best-first search* is a search technique proposed in [25], which is basically a DFS enhanced with breadth-first aspects. At intervals, all active choice points are ranked according to some internal heuristic and the *best* branch is expanded. Similarly, *generational search* introduced in [26] computes all potential new paths from a given execution and explores the *best* one,

based on a ranking of all potential new paths from all active executions.

A heuristic described in [9] deals with pruning a path exploration when the current program state is considered similar to a previously encountered state. Some research has been performed on modular test generation in order to avoid function inlining. Some of these approaches are based on function summaries [27] while others handle functions lazily [28]. The Variably Interprocedural Program Analysis heuristic [29] deals with abstracting a function call when the call stack is too high. However, the function body is not explored at all: the abstract function returns directly an unconstrained value or data store (to model arbitrary side-effects) depending on the user's choice. This prevents the procedure from exploring a single potentially deep path in callee procedures; however the resulting path predicate is too loose, and a solution is not ensured to exercise the path at runtime. Based on a similar idea, function call concretization [3, 6] is a technique in which backtracking is not allowed in the callees, but input and output values are fixed.

Grammar-based techniques have been proposed for generating complex inputs for software systems [30]. While this technique is effective, it requires a grammar to be given for the input of the program-under-test which is not always feasible. Several randomized algorithms for model checking have also been proposed. For example, *Monte Carlo Model Checking* [31] uses random walks on the state search space to give probabilistic measures on the validity of properties expressed in linear temporal logic. Statistical model checking techniques have also been proposed, which verify probabilistic models against probabilistic error bounds [32].

Path explosion due to thread interleaving in concurrent programs has been addressed in [33]. This work is based on *partial orders* methods from model checking. Randomized depth-first search and its parallel extension [34] have been proposed to improve the cost-effectiveness of state-space search techniques using parallelism. Evolutionary algorithms have also been used in the *Verisoft model checker* [35] to catch errors quickly in concurrent programs that have data inputs from a small domain. In this manner, the problem of path explosion in dynamic test generation has been addressed using several techniques, although different from ours.

Chapter 3

A Reachability-guided Search Strategy

Our first search strategy, the 'reachability-guided search' performs a reachability analysis of the control-flow graph (CFG) of the program to decide whether the current branch must be explored or not. Search strategies driven by static information from the program's CFG have been shown to enable symbolic execution systems achieve greater coverage on large software programs. For example, the CFG-directed search strategy in the test generation tool CREST chooses branches to negate based on their distance in the CFG to currently uncovered branches [8]. Our reachability-guided strategy utilizes the reachability graph of the program-under-test to explore uncovered important parts of the program. Such a reachability graph can be extracted using static analysis tools or from a specification or a high level model of the software program. This simple strategy guides the search towards the important parts of the code; these important parts are user-specified in terms of program conditionals (or nodes in the CFG).

This technique builds on the depth-first strategy, such that this additional check is performed only while backtracking through the CFG. Using this new strategy, when the test generation tool encounters a program condition, while backtracking, the tool first looks up the counter-

edge of the previously visited edge. This 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. If no new important items can be reached from the unvisited branch of the current node, then exploration from the current node stops. The search procedure then 'backtracks' to the previous/antecedent node in the CFG and this process continues until all the branches are covered. This method helps avoid re-visiting branches that do not lead to any new, important conditionals. The steps of the reachability-guided strategy are described in Algorithm 3.1.

Algorithm 3.1 *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 /* backtrack */
    CurrNode ← previous node in Path
  end if
end while
```

The benefit of the reachability-guided search can be seen in Figure 3.1. The figure shows the control-flow graph of a program that has 6 conditionals, which are depicted as the nodes of the graph.[1] Nodes $n5$ and $n6$ are terminal nodes of the graph, i.e., they do not lead to any

[1]For the sake of simplicity, the CFG shown in Figure 3.1 (and in the rest of the figures representing program CFGs) show only conditionals in the program.

more branches (e.g. *return* statements). Using the reachability-guided search strategy, we do not want to explore the branches of a condition, once it has been exercised in both directions. In this program, conditional $n6$ can be reached from conditionals $n0, n1, n2, n3$ and $n4$. Once the *true* and *false* branches of conditional $n6$ have been visited, we do not want to visit both branches of conditional $n6$ again, via other paths in the program. Using this method, we obtain complete branch coverage of the program, in a much shorter time. The original DFS-based search explores all 12 paths in the program, whereas the reachability-guided method explores only 8 paths of the program. In both cases complete branch coverage of the program is obtained.

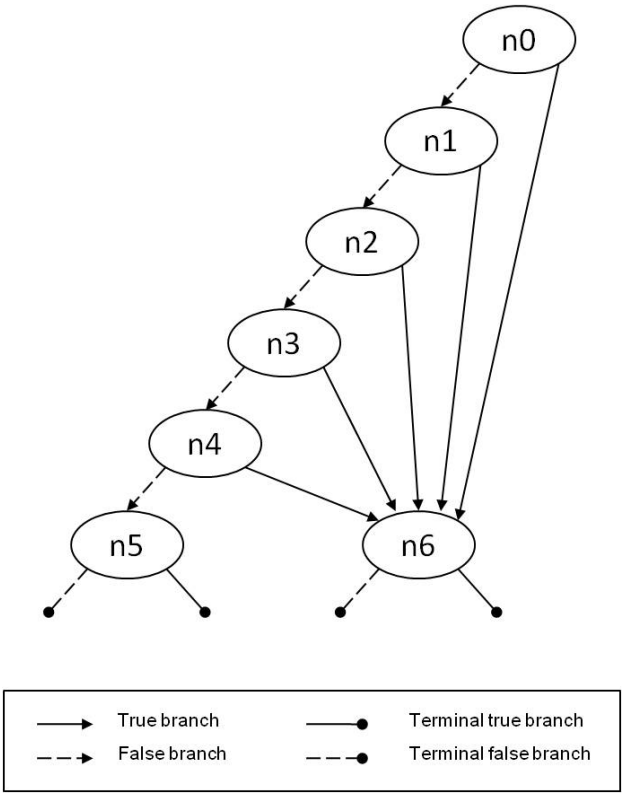

Figure 3.1: *Control-flow graph of a program to demonstrate the benefit of the reachability-guided search*

For programs with several inter-dependant function calls, the reachability-guided search strategy helps quickly achieve the goal of branch coverage. For example, the control-flow

graph shown in Figure 3.2, depicts a densely-connected control flow with several paths to reach the same node. The total number of possible paths in the graph is in the order of 100, while the number of edges is 20. When we are not interested in exercising every feasible path in the graph and are only interested in as many test cases as the number of edges in the graph (which represents complete branch coverage of the corresponding program), this search strategy helps us achieve exactly that.

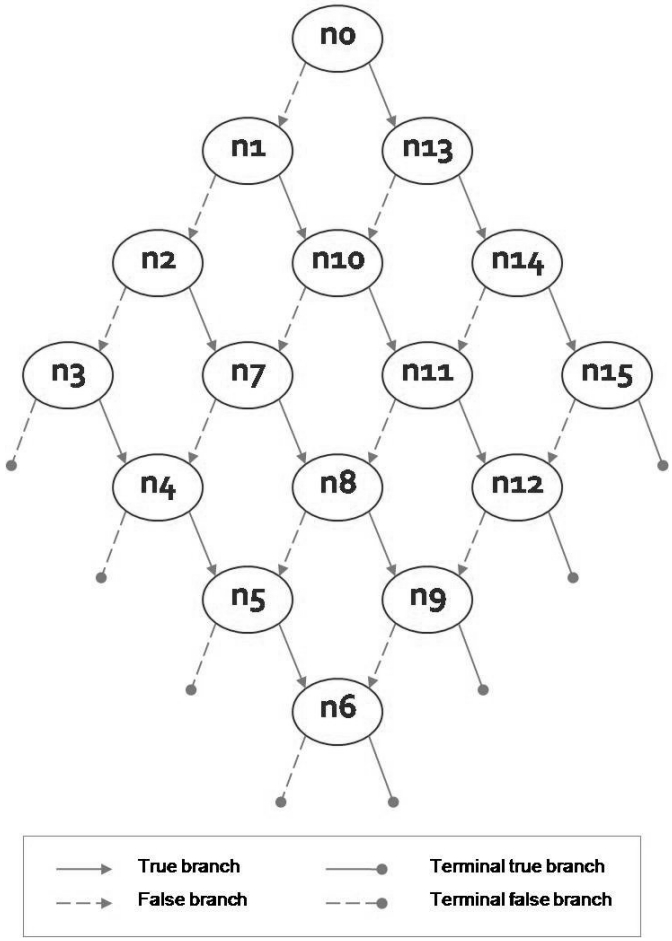

Figure 3.2: *Control-flow graph of a program to demonstrate complete branch coverage achieved using the reachability-guided strategy*

When the goal is to generate a test case to target a particular condition, just that conditional can be 'marked' important in the user-supplied list of important conditionals. For example, in Figure 3.2, say nodes $n8$ and $n9$ are not important. After exploring the path $(n0, F) \rightarrow$

$(n1, F) \rightarrow (n2, T) \rightarrow (n7, T) \rightarrow (n8, F) \rightarrow (n5, F)$ and reaching node $n8$ while backtracking, the *true* branch of $n8$ is not explored since $n8$ is not important and it does not lead to any unvisited important conditionals either (since $n9$ is not important either). On the other hand, if $n8$ was not important but $n9$ was, the *true* branch of $n8$ would be explored *until* both branches of $n9$ have been exercised.

The reachability-guided search strategy provides significant improvement in speed of test case generation by achieving complete branch coverage quickly, while exploring fewer paths. We avoid re-visiting branches that have been completely exercised previously. By using a user-supplied list of important conditionals, we generate test cases that are more relevant to the main program, without getting lost within the program's 'helper' functions. Some examples of such 'helper' functions are library functions or individual modules in the case of modular development, where each developer owns a specific module. In such a scenario, the proposed algorithm can be used to generate a system-level test that only targets the conditionals in a particular module. By performing a depth-first search of the reachability graph of the program, the proposed algorithm explores the branches in such a way as to reach the important conditionals and the search terminates once the desired conditionals have been exercised. This importance-driven strategy thus helps when we are interested in generating test cases for a subset of branches in the program.[2] Using our proposed reachability-guided strategy, we observe a significant reduction in the number of redundant test cases generated and infeasible paths explored.

3.1 Optimizations in the Search Process

In addition to the steps outlined in Algorithm 3.1, our reachability-guided strategy makes use of further optimizations to speed up the path exploration process. In this section, we describe two such lightweight optimizations that help us achieve a significant reduction in

[2]A point to note is that for our experiments detailed in Chapter 6, all conditionals are marked important for better comparison with other search techniques.

test generation time using the reachability-guided strategy.

As shown in Algorithm 3.1, the reachability analysis can be aided by supplying a list of important conditionals that need to be targeted. Sometimes, we may be more interested in a particular branch of a conditional rather than both its branches. The algorithm was thus modified to target important branches that the user specifies, instead of important conditionals. This serves the purpose of a targeted, importance-directed search in a better manner.

Our second optimization deals with the reachability analysis of the CFG of the program-under-test. In the backtracking step of the reachability-guided strategy, a depth-first search of the program CFG is done to check if any unvisited important branches can be reached from the counter-branch of the current node. If such a branch can be reached, the counter-branch is explored and the same process continues until all the branches of the CFG have been covered. This implies that, while backtracking through the CFG, a complete DFS is performed at every branch. By storing some information learned during a reachability analysis step, we can avoid performing several redundant reachability evaluations. If it is found that a particular branch, say b, does not lead to any unvisited important branches, we set a $notToVisit$ flag corresponding to that branch. The next time the reachability analysis is performed, when this branch b is encountered, we first check its $notToVisit$ flag. If this flag is set, we do not need to extend the DFS from b, since we already know that it will not lead to any branches worth exploring. In this manner, each time a branch is found, that leads to no more branches worth exploring, we set its corresponding $notToVisit$ flag. This additional step, although trivial, saves a significant number of unnecessary computations.

Chapter 4

A Conflict-driven Backtracking Strategy

In the previous chapter, we presented an approach to ensure that every new test generated covers at least a single unvisited branch. However, such an approach might still explore many infeasible paths in search of tests that can cover unvisited branches. In our experience, in several cases, significant time is spent in exploring infeasible paths in the program. We can reduce the number of infeasible paths visited, by determining the cause of the infeasibility and using this information to improve our search procedure.

As described earlier, test data to exercise a particular path in a program are generated by solving the path constraints of that path, with the help of a constraint solver. Such a solver is able to check if a constraint equation is 'satisfiable' or 'unsatisfiable'. In the case of satisfiability, it returns a solution to the constraint equation. Similarly, in the case of unsatisfiability, we can derive useful information from the solver to further improve our search process. In this section we describe our nonchronological backtracking strategy, that is similar to the search strategy found in most of the SAT solvers today [36]. We have extended this concept towards the problem of software test generation using SMT solvers. In addition to conflict-driven nonchronological backtracking, we also use 'conflict-driven learning' to avoid

revisiting paths that share the same set of conflicting assignments. This procedure is detailed in Section 4.2.

In the basic DFS based search process, the procedure backtracks chronologically to the immediately preceding decision level. However, by an analysis of the encountered conflicts, it may be possible to backtrack nonchronogically by jumping back over multiple levels in the search tree at once. In our context of path-based test generation, an infeasible path is viewed as a 'conflict', and the two terms will be used inter-changeably (depending on the context) henceforth. Let us assume that the direction-assignment at decision level β in the execution tree leads to a conflict with a previous assignment and results in an infeasible path. We call the conflicting conditionals and their direction-assignments that led to the infeasible path, a 'conflict-clause'.[1] It is worth noting that all value assignments that are made after the decision level β will force the just-identified 'conflict clause' to be unsatisfied.

A search engine that backtracks chronologically may, thus, waste a significant amount of time exploring a useless region of the search space only to discover, after much effort, that the region does not contain any satisfying assignments. For example, the conflict may be the result of two chosen assignments earlier on in the tree. In that case, it would be pointless to keep searching in a sub tree in which it is already known that there are no solutions. In contrast, our search engine jumps directly from the current decision level back to the decision level β. This multi-level backtracking is similar to the one used in SAT solvers [36] and is also known as backjumping.

An example of the benefit of this strategy can be seen in Figure 4.1. The nodes in the graph depict conditionals in a program and the solid and dotted lines represent their *true* and *false* branches, respectively. The conditionals used in this example are simple Boolean comparisons involving integers. The edges with blunt ends (without arrowheads) are used to depict that the graph shown is incomplete and that the graph continues further down those edges. Nodes without any edges leading from them depict the end of a path, i.e., an

[1] We adapt the concept of conflict-clauses used in satisfiability solvers, to our context of program path exploration.

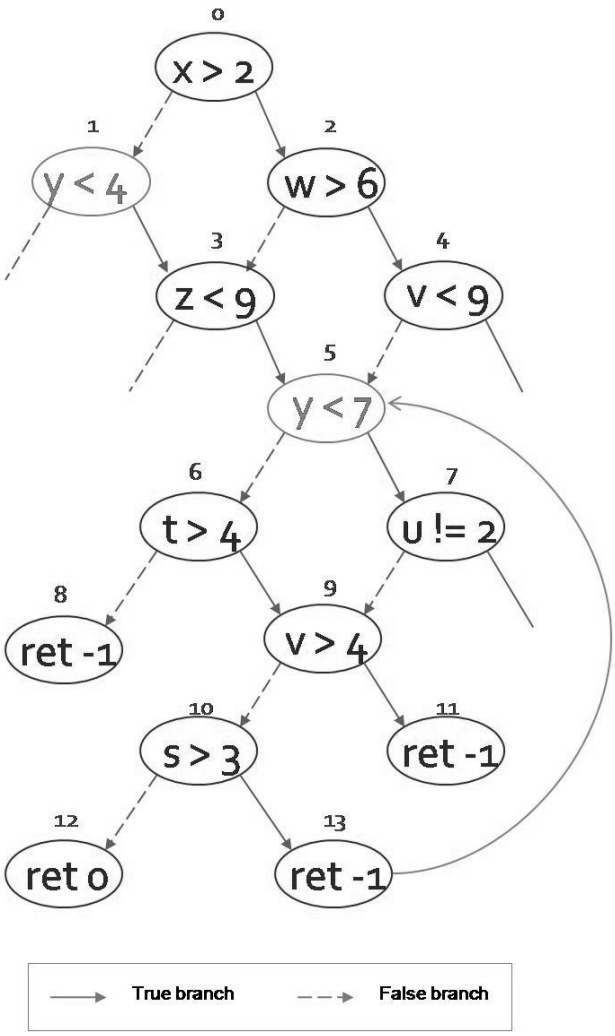

Figure 4.1: *Figure to demonstrate the conflict-driven backtracking strategy*

exit state. Let us consider the case when an infeasible path is encountered, say, $(x > 2)F \rightarrow (y < 4)T \rightarrow (z < 9)T \rightarrow (y < 7)F \rightarrow (t > 4)T \rightarrow (v > 4)F \rightarrow (s > 3)T$ in Figure 4.1. We represent this path as $(0, F) \rightarrow (1, T) \rightarrow (3, T) \rightarrow (5, F) \rightarrow (6, T) \rightarrow (9, F) \rightarrow (10, T)$. As soon as we reach the end of the path, i.e., node 13 here, we immediately derive the cause of the infeasibility. In this case, the conflict is caused by the two incompatible 'clauses' $(y < 4)T$ and $(y < 7)F$, i.e., the branch constraints $(y < 4)$ and $(y > 7)$. Since this combination of constraints can never be satisfied, all paths containing this pair of clauses will be infeasible. In the case of the basic DFS approach, the following set of infeasible paths will be explored next:

$$(0, F) \rightarrow (1, T) \rightarrow (3, T) \rightarrow (5, F) \rightarrow (6, T) \rightarrow (9, F) \rightarrow (10, F)$$

$$(0, F) \rightarrow (1, T) \rightarrow (3, T) \rightarrow (5, F) \rightarrow (6, T) \rightarrow (9, T)$$

$$(0, F) \rightarrow (1, T) \rightarrow (3, T) \rightarrow (5, F) \rightarrow (6, F)$$

In this manner, four infeasible paths are explored before we backtrack to node 5 and explore the next feasible path, say $(0, F) \rightarrow (1, T) \rightarrow (3, T) \rightarrow (5, T) \rightarrow (7, F) \rightarrow (9, T)$. In our conflict-driven backtracking strategy, we avoid visiting these infeasible paths by 'backjumping' from terminal node 13 directly to the cause of the conflict, i.e., conditional $(y < 7)$. Here, for the current path $(0, F) \rightarrow (1, T) \rightarrow (3, T) \rightarrow (5, F) \rightarrow (6, T) \rightarrow (9, F) \rightarrow (10, T)$, the current decision level (at node 10) is 6 and the level that we must backtrack to, β is 3. We then negate this conditional 5, in order to visit its *true* branch and the search then continues from this point on. Thus, for this small example, by using the conflict-driven strategy we visit only one infeasible path instead of four!

4.1 Using the Unsatisfiable Core for Conflict Analysis

In order to implement nonchronological backtracking, it is first necessary to find out the cause of the conflict, i.e., the infeasibility of the path in our case. This information can be

obtained with the help of the unsatisfiable core (or unsat core) of the SMT problem.

In the context of propositional satisfiability, it is well-known that a CNF formula is unsatisfiable if it is possible to generate an empty clause by resolution from the original clauses. The set of original clauses involved in the derivation of the empty clause is referred to as the unsatisfiable core. We extend this definition to the context of an SMT formula.

Definition 2 (Unsatisfiable Core). *Given an unsatisfiable SMT formula φ, we say that an unsatisfiable SMT formula ψ is an unsatisfiable core of φ iff $\varphi = \psi \wedge \psi'$ for some (possibly empty) SMT formula ψ'. Intuitively, ψ is a subset of the constraints in φ causing the unsatisfiability of φ.*

The SMT-LIB standard [37] describes the unsat core as a subset of the set of all assertions that the solver has determined to be unsatisfiable. With the help of the unsat core, we can find the source of unsatisfiability in an encountered infeasible path. The first step for computing an unsatisfiable core consists of identifying the clauses (either original or recorded) that were involved in the steps that led to deriving the empty clause, and thus proving unsatisfiability.

There is not much published work devoted to the computation of unsatisfiable cores in SMT. However, the problem of finding small unsatisfiable cores in SAT i.e., unsatisfiable subsets of unsatisfiable sets of clauses has been addressed by many authors in the recent years [38, 39, 40, 41]. In [41], for an instance of propositional unsatisfiability, the unsat core is computed as a byproduct of a DPLL-based proof-generation procedure. The computed unsat core is simply the collection of all the original clauses that the DPLL solver used to derive the empty clause by resolution. An algorithm to compute minimal unsat cores is presented in [42]. The technique is based on modifications of a standard DPLL engine, and works by adding some extra variables (selectors) to the original clauses, and then performing a branch-and-bound algorithm on the modified formula.

The SMT-LIB standard [37] describes a method to extract the unsat core for an SMT instance:

the solver selects from the unsat core only those formulas that have been asserted, and returns their labels. Unlabeled formulas in the unsat core are simply not reported. Unsat cores are useful for applications because they circumscribe the source of unsatisfiability in the asserted set. The labeling mechanism allows users to track only selected asserted formulas when they already know that the rest of the asserted formulas are jointly satisfiable.

Some of the current state-of-the-art SMT solvers provide ways to generate the unsat core with techniques adapted from SAT. CVCLITE [43] and a recent extension of MATHSAT [44] can compute unsatisfiable cores as a byproduct of the generation of proofs, in a way similar to that in [41]. The solver that we use, Yices [45] 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 [46]. For this purpose, each clause is represented as an assertion and each assertion in the core is identified by an assertion ID. The unsatisfiable core is then a (small) subset of the assertions that is inconsistent by itself.

When an infeasible path is encountered in our execution tree, the unsatisfiable core is extracted to find the cause of the conflict. The unsat core provided by Yices is composed of the IDs of the conflicting clauses, which are represented as assertions. The IDs of the conflict clauses are then mapped onto the conditionals where the conflicting assignments were made, (assignment in this case refers to the assignment of the direction to the condition, i.e., *true* or *false*). In the next step, instead of backtracking chronologically to the immediately preceding condition, we backtrack to a previous conditional that led to the conflict and negate it. In the simple case where the conflict is caused by two conflicting assignments, as shown in Figure 4.1, the unsat core that we obtain from Yices contains this pair of conflicting clauses $\langle (y < 4)T, (y < 7)F \rangle$. We then backjump to the previous conditional that is a part of the conflict clause pair, i.e., conditional $(y < 7)$. In the case where the cause of the conflict is more complex and the unsat core consists of more than two conflict clauses, we backtrack to the most recently visited conditional whose direction assignment led to the conflict and whose counteredge remains

unexplored. In this manner, we avoid spending a significant amount of time in searching in a sub tree in which it is already known that there are no solutions. The complete algorithm of the conflict-driven backtracking strategy is shown in Algorithm 4.1.

4.2 Conflict-driven Learning

Another pruning technique used in most state-of-the-art solvers is called *learning*. Learning extracts and memorizes information from the previously searched space to prune the search in future. Learning is achieved by adding clauses to the existing clause database. Here, we focus on learning that occurs as a consequence of conflicts created during the search process [47]. This is referred to as conflict-driven learning and from this point on, we will use the term learning only in this context.

We use conflict-driven learning in our work to further prune the search space. Each time a conflict is encountered in satisfiability solvers, the conflict analysis engine adds some clauses to the database. These learned clauses record the reasons deduced from the conflict to avoid making the same mistake in the future search. We use a similar idea for our purpose: the unsatisfiable core helps us learn valuable information from previously encountered 'conflicts' which translates to infeasible paths in our case. We use this information to avoid visiting paths that include assignments that we already know will lead to a conflict.

Every time an infeasible path is encountered during the search process, the unsatisfiable core (not necessarily minimal) is extracted. This information is translated into a small set of conditionals and their branch assignments that led to the conflict. In the context of SAT solvers, conflict clauses are composed of a combination of literals, where each literal denotes a variable and its assignment. The combination of these variable assignments is known to cause a conflict. In our context, a literal denotes a conditional along with its direction assignment (i.e., *true* or *false*). Consequently a conflict clause represents a subset of the branch assignments that led to an infeasible path. In our example depicted in Figure 4,

Algorithm 4.1 *Conflict-driven backtracking-based 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
    if (branch assignment = false) violates any learned conflict clauses then
      transition ← true branch of CurrNode
    else
      transition ← false branch of CurrNode
    end if
    if transition can reach any unvisited important branches 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
          extract unsat core and map core to branches
          blevel ← compute backtrack level
          update conflict clause database
          record Path as infeasible
        end if
      end if
    end if
  else if covered only one branch of CurrNode then
    /* symmetric case for counter-branch */
  else
    if Path was infeasible then
      remove all nodes following blevel from Path
      CurrNode ← node in blevel /* backjump */
    else
      remove CurrNode from Path
      CurrNode ← previous node in Path /* backtrack */
    end if
  end if
end while
```

the conflict clause learned is $\langle(1, T), (5, F)\rangle$ This clause is then stored in a conflict clause database, in a manner similar to those maintained in SAT solvers.

In the following iterations of our search, when we select a new branch to explore, we first check if the chosen direction assignment (i.e., *true* or *false*) for any branch in the path contains any of the learned conflict clauses. If such a violation is found, we flip the faulty assignments until we obtain a path, free of any conflicting assignments, or backtrack further up our search tree. This additional step of learning helps us avoid re-visiting infeasible paths and further reduces the run-time.

4.3 Completeness of the Algorithm

Further, we prove that our nonchronological strategy is complete and does not result in any loss of branch coverage. We base this inference on the proof of completeness of the GRASP algorithm [36], since our nonchronological backtracking strategy is based on this strategy. For reference, we reproduce a part of the discussion here, with repect to our strategy.

Lemma 1. *Let β be the backtracking decision level computed by the conflict analysis engine. Furthermore, let A_β denote the partial path assignment containing all the conditionals and their direction assignments with decision levels no greater than β. In this scenario, a solution cannot be found for any path assignment A such that $A \supset A_\beta$.*

In other words, a solution cannot be found for any path, with path predicate P such that $P \supset P_\beta$, where P_β is the path predicate corresponding to A_β.

Corollary 1. *Let α be the current decision level and β be the computed backtracking decision level. In this scenario, a feasible path to the current branch of the conditional at level α cannot be found until the search process backtracks to decision level β.*

Definition 3 (Completeness). *In our context of path exploration, a search strategy is said*

to be complete if, for every branch, a feasible path (i.e., solution) will be found if at least one such feasible path exists.

Theorem. *The nonchronological backtracking search algorithm is complete.*

Proof. It is well known that the depth-first strategy, with chronological backtracking is a complete search procedure [48]. Backtracking search extends partial assignments until the depth of the search tree has been reached. In the event of an inconsistent path assignment, i.e., an infeasible path being encountered, the most recent decision assignment yet untried is considered and the search proceeds. Hence, if a solution exists, it will eventually be found. Given these facts, we only need to prove that nonchronological backtracks don't jump over partial assignments that can be extended to feasible path assignments. Let us suppose that the current decision level is α and the computed backtracking decision level is β. Then, by Lemma 1 and Corollary 1, we can conclude that a feasible path for the current branch cannot be found by extending any partial path assignment defined by decision levels greater than β on the current decision path, i.e., all path assignments $\{A_\gamma \mid \beta < \gamma \leq \alpha\}$ are infeasible. It then follows that if a feasible path to the current branch exists, it will eventually be enumerated and, consequently, the algorithm is complete.

Chapter 5

Error-targeted Search Strategies

The previous two chapters deal with finding ways to quickly achieve complete branch coverage of the program-under-test. We did not concern ourselves with the actual reason for targeting complete branch coverage - the discovery of program errors or 'bugs'. However, bug-discovery is the main aim of test generation. By generating tests to exercise every branch of the program-under-test, we hope to uncover bugs hidden in the program. Sometimes, we might be more interested in catching bugs quickly, while at the same time, achieving complete branch coverage of the program. Keeping this in mind, we propose strategies aimed at finding bugs quickly, in this chapter.

5.1 An Assertion-directed Search Strategy

An abort statement is used to denote a failure exit state of the program, leading to abnormal program termination. In C, the function generates the SIGABRT signal, which by default causes the program to terminate returning an unsuccessful termination error code to the host environment. This error exit state should not occur during correct execution of the program.

An assertion is a predicate placed in a program to indicate that the developer thinks that

the predicate is always true at that point. If this expression evaluates to 0, this causes an assertion failure that terminates the program. In C, a message is written to the standard error device and **abort()** is called, terminating the program execution.

Programmers use assertions to reason about program correctness. An assertion may be used to verify that an assumption made by the programmer during the implementation of the program remains valid when the program is executed. For example, a precondition an assertion placed at the beginning of a section of code determines the set of states under which the programmer expects the code to execute. A postcondition placed at the end describes the expected state at the end of execution. If the assertion does not hold, an error can be reported.

The assumption that a particular variable will have one of a small number of values is an invariant, which can be checked with an assertion, as shown in Figure 5.1. The assertion in the program may fail if i is negative, as the % operator is not a true modulus operator, but computes the remainder, which may be negative. Here, the programmer has assumed that i is non-negative, so that the remainder of a division with 3 will always be 0, 1 or 2. The assertion makes this assumption explicit if i does have a negative value, the program may have a bug.

```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else {
    assert (i % 3 == 2);
    ...
}
```

Figure 5.1: *A C program to demonstrate the usage of assertions*

Assertions are often used to document logically impossible situations and discover programming errors if the "impossible" occurs, then something fundamental is clearly wrong. Assertions are also placed at points the execution is not supposed to reach. For example, in

languages such as C, C++, and Java, assertions could be placed at the default clause of a switch statement with no default case. The absence of a default case typically indicates that a programmer believes that one of the cases will always be executed. By using an assertion here, any case which the programmer does not handle intentionally will raise an error and the program will abort rather than silently continuing in an erroneous state. **abort()** statements and assertions are inserted by the programmer because he/she thinks that there is a *possibility* of an error at that location in the program.

In this section, we target these programmer-inserted assertions and **abort()** statements. In our previous strategies, we mainly dealt with modifying the manner in which we backtrack during the depth-first search of the CFG of the program. The first path that was being explored was the 'all-false' path, i.e., the *false* branch being selected for each condition, as shown in Algorithms 2.1, 3.1, 4.1. In this section, we use strategies to intelligently select the initial paths to be explored.

A useful method to select the initial paths to explore is choosing those paths that lead to user-inserted assertions and **abort()** statements. In this assertion-directed strategy, we first explore those branches that lead to possible 'failure states', such as **abort** statements and assertions. Once these 'failure states' have been reached, the rest of the CFG is explored until we obtain complete branch coverage using the reachability-guided search, which is set as the default strategy. This initial-path selection can also be used in conjunction with the other search strategies described previously - traditional DFS or the conflict-driven backtracking strategy. In this manner, we target the assertions in the program, while achieving complete branch coverage.

Figure 5.2 depicts the control flow graph of a procedure or function in a program. Each node in the graph represents a conditional in the program and its two edges denote the *true* and *false* branches of the conditional. The terminal nodes *ret 0* and *ret 1* represent valid exit points of the procedure, like *return* statements. The *abrt* node denotes an **abort()** statement or a failure state in the program. Encountering this state during program execution

represents an error in the program. We can use the assertion-directed strategy to detect this bug quickly.

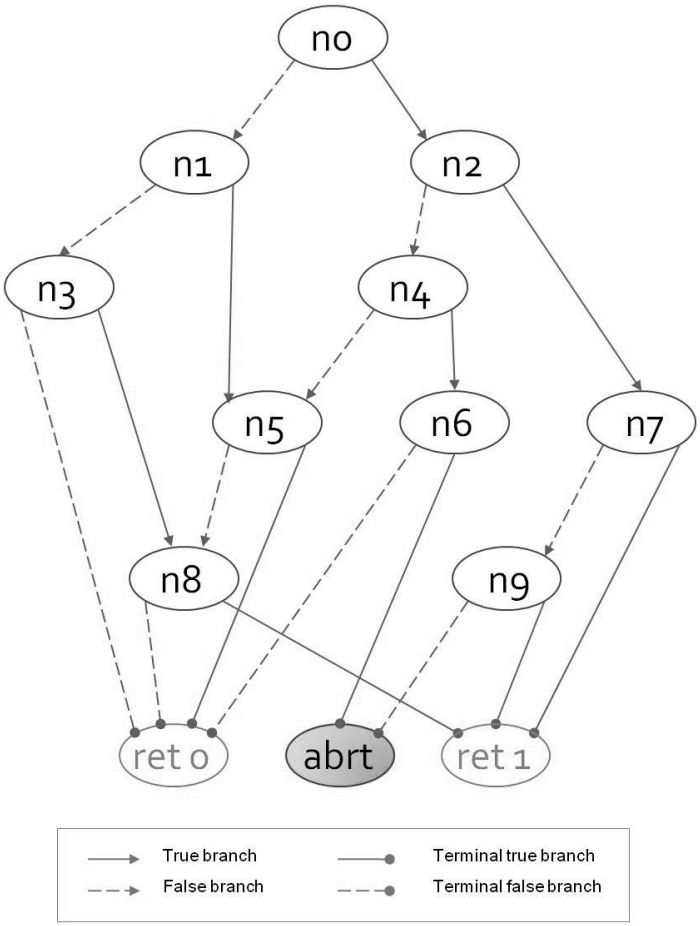

Figure 5.2: *A representation of a program CFG to demonstrate the assertion-directed strategy*

Using the reachability-guided strategy, the default initial 'all-false' path is explored: $(n0, F) \rightarrow (n1, F) \rightarrow (n3, F)$. The depth-first search continues from this point on, taking into account the reachability of unvisited conditionals. By selecting the *false* branch as the first edge to visit at each encountered condition, the abort statement is reached only in the 8th iteration. Using the assertion-directed strategy, the program-under-test is parsed to determine the number of assertions/abort statements in the program. A reachability analysis is then performed to select the paths that reach conditionals that lead to the **abort()** state. The

strategy first explores the two paths that lead the the abort statement:

$$(n0, T) \rightarrow (n2, T) \rightarrow (n7, F) \rightarrow (n9, F)$$

$$(n0, T) \rightarrow (n2, F) \rightarrow (n4, T) \rightarrow (n6, T)$$

The search procedure further continues exploring the rest of the graph in the depth-first manner along with the reachability analysis done at each step to avoid revisiting covered branches. In this manner, the bug in the program (the **abort()** statement) is found in the first iteration of the testing process.

In many cases, it has been observed that the intelligent selection of the first path that is explored can lead to a significant improvement in the effectiveness of test generation. By our proposed strategy, in the event of an abnormal program termination or a failed assertion, we reach these failure states early in the testing cycle. This method shows significant savings when testing large programs where, when using the traditional selection of the first-explored path, a bug is found only after several iterations.

5.2 A Detectability-based Heuristic Targeting Hard-to-find Errors

In the assertion-directed search strategy and the importance-directed reachability-guided strategy, we used information that is either inserted in the program or supplied externally by the programmer during testing. In the previous section, we outlined a method to target pre-inserted assertions and abort statements when selecting paths to explore in the program-under-test. In most real-life cases however, we are not aware of the location of bugs in the program being tested.

In the more common situation where we are only given a program to be tested, we are interested in finding a way to target program errors quickly. We do not know the location

of program errors, but we can extract useful information from the static control-flow graph of the program to guide our search process. In this section, we outline one such strategy to possibly make it easier to find hard-to-detect bugs.

In the context of testing for software security vulnerabilities, McGraw [49] states that testing for such problems is complicated by the fact that they often exist in hard-to-reach states or crop up in unusual circumstances. In the symbolic execution-based bug-finding tool EXE as well, the search server picks the branch at the line of code run the fewest number of times, instead of simply picking a random branch to follow [25]. We base our strategy on the assumption that hard-to-detect bugs are those that are hidden in obscure parts of the program, i.e., in paths that are less likely to be exercised. Bugs in branches that many paths lead to, are not hard-to-reach - they will be exercised through one path or or the other. The bugs that can be reached only through a few paths in the program are the ones that are hard-to-detect. After all, in order to answer the question "Does program P have bug X?", we first need to be able to answer the question "Can program P reach state X?". Several metrics can be used to define 'hard-to-detect' software program errors; in this work, we use the term 'hard-to-detect' errors to mean those errors that can be found only in hard-to-reach parts of the program-under-test.

Based on the description of 'hard-to-detect' errors from the previous paragraph, we can determine the probable locations of such errors in hard-to-reach branches of the program-under-test. These hard-to-reach branches in turn can be found by evaluating the number of paths that lead to each node in the execution tree, that represent the conditionals in the program. This can be restated as the well-known problem of counting paths in a directed acyclic graph (DAG). We have described earlier that there are often exponentially large number of paths in the program execution tree (see Section 2.5). However, it is possible to efficiently count the number of paths using the linear-time algorithm outlined in Algorithm 5.1.

Based on the number of paths that lead to each condition, each branch in the execution tree

Algorithm 5.1 *Counting paths in a DAG from source, S to destination, D*

```
path counts of all vertices N(U) ← 0;
path count of source node, N(S) ← 1
topologically sort the vertices
while node, D is not reached do
  for each vertex U after S in topological order do
    for each neighbor V of U do
      N(V) ← N(V) + N(U)
    end for
  end for
end while
return N(D)
```

is assigned a weight, that we call its *detectability weight*. This *detectability weight* of a branch, b, is calculated as the ratio of the number of paths leading to b, to the total number of paths in the CFG. During the traversal of the program's CFG, the search process selects those branches in the path, with a lower *detectability weight*.

Detectability weight DW_b of a branch b, is defined as: $DW_b = \frac{N(i)}{P}$, where,

i is the consequent node of b, i.e., the node that b leads to,

$N(i)$ is the path-count of node i and

P is the total number of paths in the control flow graph.

At every choice point or condition, encountered in the program, the branch with the lower *detectability weight* is chosen, instead of the default *false* branch. If the weights of both branches of a node are the same, one of them is chosen at random to be explored first. This strategy is explained with the help of the control flow graph depicted in Figure 5.3. The graph depicted in this figure is identical to the one shown in Figure 5.2, except for the addition of the path-counts of the nodes in the graph. These path-counts are shown to the left of each node in the graph.

Based on the path-counts of each node, the *detectability weights* of each branch are computed. During the initial path selection, since both nodes $n1$ and $n2$ have the same count, let us assume that the *true* branch is selected first. The two successor nodes of $n2$ - $n4$ and $n7$,

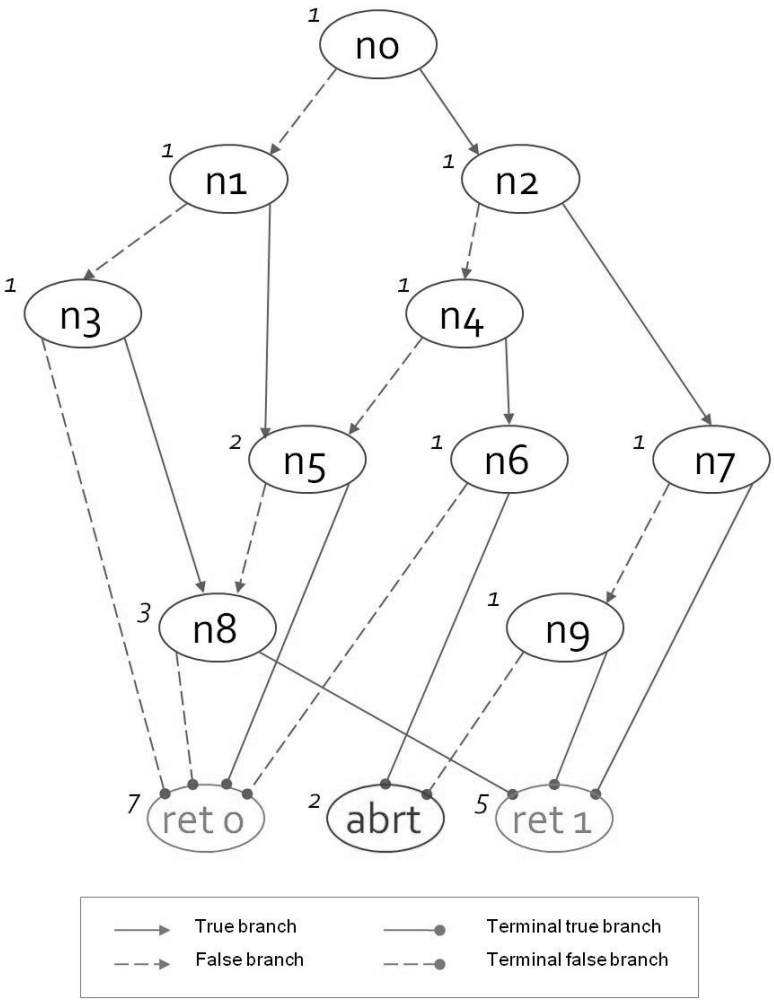

Figure 5.3: *A representation of a program CFG to demonstrate the detectability-based search strategy*

again have the same path-counts and consequently, the same detectability weights; here, let us assume that the *false* branch is explored first, so as to reach node $n4$. At this point, the *true* branch of $n4$ is selected since it has a lower weight of $\frac{1}{14} = 0.0714$, compared to the false branch, whose weight is $\frac{2}{14} = 0.1429$. Similarly, at node $n6$, with a weight of 0.1429, the *true* branch is chosen over the *false* branch, which has a weight of 0.5. The path thus explored is $(n0, T) \rightarrow (n2, F) \rightarrow (n4, T) \rightarrow (n6, T)$, which leads to the **abort()** statement. The path exploration continues thereafter, in the depth-first manner, [1] selecting the branch with a lower *detectability weight* at every newly visited condition.

In the strategies described in Chapters 3 and 4, the first branch to be explored at each node is chosen by default, without any analysis being done. This often leads to the exploration being 'stuck' in particularly 'dense portions' of the CFG. Using the detectability-based heuristic, at every newly visited condition, the branch that leads to fewer paths is selected for exploration. In this manner, the traversal of the CFG is biased towards hard-to-reach parts of the program, which are less likely to be exercised. By doing so, we aim to provide the test generation tool with some hints about the possible locations of hidden bugs in the program. Consequently, we could find bugs hidden in these obscure branches, earlier in the testing process.

[1] This refers to DFS in conjunction with the reachability analysis

Chapter 6

Implementation and Experimental Evaluation

In this chapter, we first describe the implementation of our proposed search strategies in our test generation framework and then measure the effectiveness of our strategies on testcases based on real world benchmarks. We evaluated our implementation on examples derived from Test Programs that are used in the high volume manufacturing of Intel's semiconductor devices.

6.1 Some Implementation Details

In this section we first describe the implementation of our search strategies in our test generation framework. For our initial experiments, we implemented our algorithm on a test generation tool, driven by symbolic execution. This tool is built on the lines of the DART [6] and CUTE [3] frameworks. It is written in the functional language, OCaml and is composed of three main components - an instrumentation tool, a symbolic execution library, a path exploration framework. CIL [21], an OCaml framework for parsing, transforming and

analyzing C code, is used to instrument the program-under-test for symbolic execution. The library deals with symbolic execution of the program being tested. The path exploration framework deals with the exploration of the execution tree of the program. The path constraints are solved with the help of the Yices SMT solver [45].

6.1.1 An Alternate Test Generation Framework

For experiments to tackle the path explosion problem, we are mainly interested in the path exploration framework of the test generation tool. In order to work on this problem independently, we implemented our search strategies on a stand-alone path exploration framework without the overhead of program instrumentation and symbolic execution. The proposed framework is written in C++ and makes use of the parsing utilities Lex and Yacc. The SMT solver Yices is used as the constraint solver.

The proposed test generator takes as input a control flow graph of a program, represented in the form of a finite state machine.[1] The test generator selects a path to explore by accumulating the branch constraints of each state (or node in the graph) along the path and provides this conjunction of constraints to the SMT solver. If the solver is able to find a solution, the obtained solution serves as the test vector for the path and if no solution is found, the path is recorded as an infeasible path. It then selects the next path to explore, based on the search strategy selected. We implemented the strategies described in this thesis in this framework. Based on the user's selection, the algorithm can perform a path-exhaustive, reachability-based, the nonchronological backtracking-based search or an error-directed search.

Our testing framework can be used for testing any model that captures the behavior of a system. This framework is thus programming language independent as well, as long as the intent of the program can be expressed in the form of a control-flow graph and the conditionals in the program can be expressed using the available SMT theories. Programs written in any

[1] A point to note is that in order to mimic code produced by CIL, this CFG must be acyclic with all loops unrolled, and all compound conditionals converted to simple conditionals, prior to test generation.

C-like programming language and even in register-transfer-level (RTL) form can be modeled in the form of a finite state machine. In this manner, we concern ourselves only with the path-selection and constraint solving aspects of dynamic test generation, without dealing with the aspects related to symbolic execution of the program-under-test. For the purpose of this work, we made use of these test generation frameworks. Our search framework can be integrated onto any of the constraint-based testing tools.

6.2 Experimental Setup

In this section, we measure the effectiveness of our proposed search strategies on testcases based on real world benchmarks. We evaluated our implementation on examples derived from Test Programs that are used in the high volume manufacturing of Intel's semiconductor devices. The results obtained are promising and show that the nonchronological backtracking technique, along with the reachability analysis, significantly reduces the number of tests required to achieve the goal of branch coverage of the program. We performed our experiments on a dual-core 2.8GHz Intel Core 2 Duo machine with 2.9 GB of RAM.

6.3 Verification of Test Programs

Test Programs are software entities written on top of a tester operating system and used to control the tester hardware. They determine whether a Silicon device is defect-free and also its product category. A bug in the Test Program software can result in defective units being shipped to customers or working units being discarded or binning of the units into incorrect product categories. These Test Programs are written in C and are loop-free, but control-intensive. The number of paths in Test Programs is often in the order of millions. Current Test Program validation techniques involve running a large number of hardware units on testers and exhaustive manual inspections. Such techniques are expensive (hardware and

material costs), effort intensive and increase the time to market for a product. We applied the approach proposed in this thesis towards this problem of Test Program validation to automatically generate test cases so that the Test Program software can be comprehensively and efficiently validated in an offline environment.

6.4 Results

In this section, we present experimental results of evaluating our search strategies on test-cases derived from Intel's Test Programs. We compare two of our proposed strategies, the reachability-guided strategy and the conflict-driven nonchronological backtracking strategy with the base DFS strategy, which we call the path-exhaustive strategy. An important point to note is that, since both of our strategies are complementary, we implemented our nonchronological backtracking strategy, as an enhancement of the reachability strategy. Thus, in addition to performing the reachability analysis, the nonchronological backtracking strategy performs conflict analysis to determine the level to backtrack to and uses conflict-driven learning to avoid revisiting already explored infeasible paths. In the remainder of this thesis, we refer to this combined strategy as the conflict-driven backtracking-based strategy. Next, we evaluate our proposed search strategies against other efficient search strategies proposed in recent work. We compare our conflict-driven backtracking-based search strategy with two search strategies in CREST, an open-source test generation tool for C [8]. Finally, we evaluate the effectiveness of our assertion-directed and *detectability*-based search strategies against the reachability-based strategy, in targeting program errors quickly.

The results obtained for the three strategies: path-exhaustive search (PE), reachability-guided search (RG) and conflict-driven backtracking-based search (RG+CB), are shown in Table 6.1. In this table, for each test case (which is derived from a Test Program), **# conds** denotes the number of distinct conditionals in the program model, where each conditional leads to a *true* and *false transition*. Based on the outcome of the condition, each *transition* leads to either

another such state or a terminal state in the model. In Table 6.1, **Paths exp.** denotes the number of feasible paths explored and **Inf. paths** indicates the number of infeasible paths explored. **Time (s)** is the CPU run time in seconds and **Speedup** indicates the speedup of the RG+CB method over the RG method. The table entries without results listed (denoted by –), represent instances for which the corresponding experimental run timed-out (the time-out was set to 2 hours). In each of the experiments, statistics are shown for complete branch coverage of the model.

Table 6.1: *Comparison of the path-exhaustive, reachability-guided and conflict-driven backtracking strategies*

Test-case	# conds	Path-exhaustive (PE)			Reachability-guided (RG)			Conflict-driven backtracking (RG+CB)			
		Paths exp.	Inf. paths	Time (s)	Paths exp.	Inf. paths	Time (s)	Paths exp.	Inf. paths	Time (s)	Speedup
tc1	17	6,816	576	5.58	32	12	0.14	19	1	0.07	2.0
tc2	20	12,944	33,136	23.48	1640	5574	5.68	18	9	0.06	94.67
tc3	28	3,238,240	73,760	2,596.01	28	137	0.53	28	3	0.16	3.31
tc5	30	4,239,360	0	2,701.85	31	0	0.12	31	0	0.12	1.0
tc18	34	–	–	–	407	192	4.14	34	1	0.28	14.79
tc6	39	–	–	–	9,590	9,568	73.67	41	1	0.28	263.11
tc19	39	–	–	–	35	843,264	525.41	35	2	0.18	2,918.94
tc7	42	–	–	–	–	–	–	42	4	0.25	N/A
tc20	42	1,716	0	0.93	43	0	0.06	43	0	0.06	1.0
tc8	47	–	–	–	3,868	3,840	55.91	53	4	0.53	105.49
tc9	61	–	–	–	127	50	2.59	62	3	0.71	3.65
tc22	84	1,159,488	0	1,026.91	85	0	0.21	85	0	0.21	1.0
tc23	143	–	–	–	–	–	–	194	8	5.84	N/A
tc10	176	–	–	–	–	–	–	180	2	8.02	N/A
tc11	180	–	–	–	268	126	26.9	191	32	10.95	2.46
tc24	210	–	–	–	211	0	0.82	211	0	0.81	1.00
tc13	445	–	–	–	622	182	193.34	467	28	114.98	1.68

'-' indicates time-out in 2 hours.

From these results, we can see that the RG strategy greatly outperforms the PE strategy for all cases. For most testcases with more than 30 distinct conditionals (except testcases without any infeasible paths), the path-exhaustive strategy results in a time-out. We also observe that, unsurprisingly, the RG and RG+CB strategies give the same results for testcases with no infeasible paths in the program model. From this we can infer that, as long as there are no infeasible paths in the model, the RG strategy performs well enough. However, as the size of the model increases and the number of infeasible paths in the program increases, the RG strategy takes significantly longer to complete and in some cases, even results in a time-out. In addition, we see that for some testcases, the number of feasible paths explored is also lower using the RG+CB strategy than the RG strategy. This is attributed to the fact that the reachability analysis is only performed while backtracking through the current path and not whenever a new path is selected. Also, every time a new path is selected, the default branch that is selected for every subsequent conditional is the *false* branch. This leads to additional feasible paths being explored using the RG strategy. For example, for tc18 with 34 conditionals, the path-exhaustive method timed out in 2 hours. The reachability-guided method needed to explore 407 paths to complete the search. The total time taken is less than 5 seconds! Next, in the conflict-driven backtrack method, only 34 paths needed to be explored. Among the 192 infeasible paths, only 1 needed to be considered, as the rest could be readily blocked from the added conflict clauses. The total time taken was less than a second - this is more than 14× speedup over the RG strategy!

Table 6.2 shows the results for experiments with the RG and RG+CB strategies, for varying number of 'conflicts' introduced into the model. In this table, for each test case (derived from a Test Program), **# conds** represents the number of distinct conditionals in the program model, as in Table 6.1. **# conflicts** denotes the number of inconsistencies introduced in the model for the purpose of evaluation. These inconsistencies in turn, lead to infeasible paths in the program model. **Paths exp.**, **Inf. paths** and **Time (s)** retain the same meanings as in Table 6.1. **Largest jump** indicates the size of the largest nonchronological backtrack.

We can see from Table 6.2 that, when there are no inconsistencies in the model, the two

Table 6.2: *Statistics of running experiments with reachability-guided and conflict-driven strategies, for varying number of conflicts*

Test-case	# conds	# conflicts	Reachability-guided (RG)			Conflict-driven backtracking (RG+CB)			
			Paths exp.	Inf. paths	Time (s)	Paths exp.	Inf. paths	Largest jump	Time (s)
tc2	20	0	19	0	0.06	19	0	-	0.05
		2	30	9	0.09	21	2	9	0.05
		4	62	1,467	1.16	20	4	12	0.06
		8	1,640	5,574	5.68	18	9	13	0.06
tc15	36	0	38	0	0.14	38	0	-	0.23
		4	56	32	0.53	39	4	9	0.24
		6	134	225	1.03	39	6	9	0.27
		7	2,119	2,654	5.49	40	7	9	0.27
tc16	47	0	48	0	0.34	48	0	-	0.34
		3	65	27	0.54	51	3	5	0.34
		6	2,008	9,115	26.92	48	5	16	0.37
tc9	61	0	62	0	0.72	62	0	-	0.7
		1	127	50	2.59	62	3	12	0.71

'-' indicates that no back-jumping was performed.

strategies RG and RG+CB perform identically, as expected. This is because the conflict analysis engine is not invoked unless an infeasible path is encountered. The important point to note in this table is that, as the number of conflicts increases, the number of infeasible paths explored under the RG strategy markedly increases and consequently, so does the run-time. We can also see that the number of feasible paths explored using the RG strategy is much larger in many cases, as previously explained with reference to Table 6.1. In the case of the RG+CB strategy, with the help of the intelligence of the conflict analysis engine, infeasible paths in the model once explored, are not visited again. For example, for tc2 with 20 conditionals, both the reachability-guided method and the conflict-driven backtrack method achieved complete branch coverage in less than 0.1 seconds, by exploring just 19 feasible paths, when there were no conflicts in the model. However, when 8 conflicts were introduced in the same model, the RG method took 5.68 seconds to complete, while exploring 1640 feasible paths and 5574 infeasible paths. Whereas, the RG+CB method needed to explore only 18 paths and 9 infeasible paths for complete coverage. The time taken by the RG+CB method was only 0.06 seconds, more than 90× speedup over the RG method. From

these results we can infer not only that the RG+CB strategy significantly outperform the PE and RG strategies, but also that this strategy is especially beneficial when the program model we are testing has several inherent inconsistencies.

In Tables 6.1 and 6.2, we compared our search strategies with the default strategy in most path-based testing tools, the DFS-based path-exhaustive procedure. We also evaluated our proposed search strategies against other efficient search strategies proposed in recent work. In Table 6.3, we compare our conflict-driven backtracking-based (RG+CB) search strategy with two search strategies in CREST, an open-source test generation tool for C [8]. The table contains results for comparison of our RG+CB strategy with CREST's *uniform-random* and *CFG-directed* search strategies. The *uniform random* strategy samples the path space of a program rather than the input space, by generating paths uniformly at random, and the *CFG-directed* search attempts to increase branch coverage of a program by driving the execution down *short* static paths to currently uncovered branches. These two strategies are represented in Table 6.3 as **CREST (UR)** and **CREST (CFG-D)** respectively. For each testcase, **# conds** represents the number of conditionals in the program and **# iters** represents the number of iterations or paths explored (feasible and infeasible) to obtain complete branch coverage. The last two columns denote the speedup obtained by the RG+CB method over UR and CFG-D, respectively. Both these methods used in CREST use heuristics to select new branches to explore, due to which, each CREST run may result in different paths being explored and consequently, different statistics. The statistics shown here were obtained for each testcase by averaging the results obtained from 10 separate runs. Another point to note is that the statistics reported for CREST run-time do not include the time taken for instrumenting the program for symbolic execution.

From the statistics in Table 6.3, we can see that our RG+CB strategy greatly outperforms CREST's uniform-random search procedure and in most cases, performs better than the CFG-directed search procedure. For example, in the case of tc18, a program with 34 conditionals, the CREST (UR) method required 196 iterations to achieve complete branch coverage and the CREST (CFG-D) method required 176 iterations, whereas our method required just 35

Table 6.3: *Comparison of conflict-driven backtracking strategy and CREST search strategies*

Test-case	# conds	CREST (UR)		CREST (CFG-D)		RG+CB		Speedup of RG+CB	
		# iters	Time (s)	# iters	Time (s)	# iters	Time (s)	v/s UR	v/s CFG-D
tc2	20	74	1.77	19	0.39	20	0.07	25.33	5.53
tc17	27	89	2.25	33	0.73	28	0.16	14.04	4.57
tc18	34	196	4.96	**176**	3.45	**35**	0.28	14.18	**12.32**
tc19	34	118	2.62	39	0.76	37	0.18	14.57	4.23
tc14	37	124	3.21	43	0.82	43	0.24	13.37	3.4
tc20	42	430	9.51	**62**	0.69	**43**	0.06	158.42	**11.53**
tc9	61	382	9.29	67	1.36	65	0.71	13.08	1.91
tc21	73	432	9.98	106	2.08	80	1.51	6.61	1.37
tc22	84	4,779	32.64	**116**	2.24	**85**	0.21	153.95	**10.56**
tc23	143	1,058	29.29	159	3.77	194	5.52	5.31	0.68
tc24	210	53,524	393.76	293	5.87	211	0.83	475.33	7.08

iterations. The RG+CB method resulted in a speedup of 14.18 over the CREST (UR) method and a speedup of 12.32 over the CREST (CFG-D) method. From these results, we can see that for nearly all programs, our proposed search strategy performs significantly better than the current search strategies in CREST. The search strategies used in CREST to quickly obtain branch coverage are based on heuristics to select new branches and discard previously visited paths. These heuristics are found to be insufficient when the program being tested has several infeasible paths or several long paths within a sub-tree of its CFG. Our proposed strategy, on the other hand, deals with these problems efficiently. The combination of the reachability and conflict analyses intelligently guide the search process, avoiding numerous infeasible paths and achieving complete branch coverage quickly.

Next, we evaluate our strategies aimed at targeting program errors quickly; our assertion-directed strategy that first explores those paths that lead to assertions or abort statements and then proceeds with the exploration of the rest of the CFG, and our *detectability*-based heuristic that targets hard-to-reach parts of the program. The plots shown in Figure 6.1 compare the number of iterations required to reach the assertions and abort statements in the program, using the reachability-guided (RG) strategy, the assertion-directed (AD) strategy and the *detectability*-based heuristic (DB).[2] From these plots, we can see that by using the

[2]For these experiments, we used programs without any infeasible paths, for which both RG and CB

AD strategy, we can quickly determine if any of the programmer-inserted assertions can be violated or if the program may abnormally terminate by reaching an undesirable 'failure state', such as an **abort()**. For example, in the case of tc26, by using the RG strategy, we reach the 3 assertions in the program at iterations 27, 30 and 31, respectively. For the same program, using the AD strategy, we discover these 3 assertion statements within the first 5 iterations.

The assertion-directed strategy is very effective when we know precisely what kind of errors we expect to find in the program-under-test. For example, in our experiments, we have considered two possible program failures: assertion failures and abort statements. When we are not aware of what kind of bugs to expect or their possible locations in the program-under-test, we can only use a metric to target the probable locations of bugs in the program. We evaluate the effectiveness of the DB strategy in finding assertion failures and abort states in the same test programs used to evaluate the assertion-directed strategy. From the plots in Figure 6.1, we can see that in most cases, the *detectability*-based heuristic is able to detect bugs in the programs quickly. We also observe that, in all cases, this strategy is able to find the errors faster than the reachability-guided strategy. For instance, for tc26, the RG strategy reaches the 'failure states' in the program at iterations 27, 30 and 31, whereas, the DB strategy reaches these same states much earlier: at iterations 3, 5 and 17. From these results, we can see that, by using a metric to target bugs in the program, in conjunction with our RG+CB strategy, we can catch errors early, while still achieving complete branch coverage quickly.

strategies perform identically.

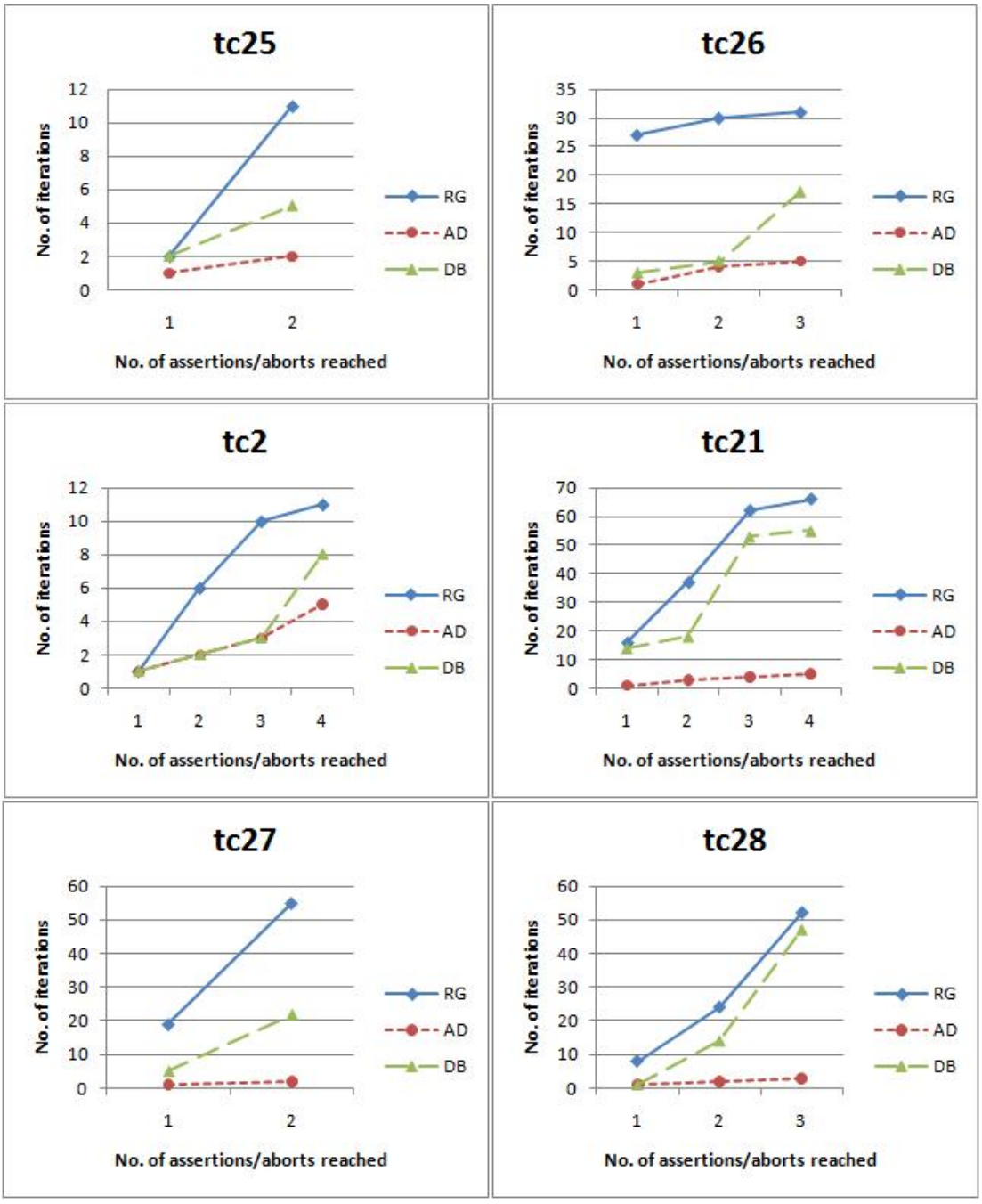

Figure 6.1: *Results obtained using the error-directed search strategies*

Chapter 7

Conclusion

Constraint-based test generation is a promising technique to automatically generate tests with high coverage. However, it suffers from the major bottleneck of path explosion in large applications. This work introduces intelligent search strategies to help solve this scalability challenge. We measured the effectiveness of our implementation by applying it to examples derived from Test Programs used for high-volume manufaturing of integrated circuits. We also compare our strategies with other efficient search strategies proposed in recent work.

Our first proposed search strategy, the reachability-guided strategy guides the search process towards unvisited important parts of the program-under-test. Our experiments show that the reachability-guided strategy can greatly reduce the number of tests required to achieve branch coverage quickly. The conflict-driven backtracking-based strategy uses conflict analysis to determine the cause of an encountered path being declared infeasible. Subsequently, this strategy utilizes conflict-driven backjumping to backtrack over multiple levels of the graph to the cause of the infeasibility. Our conflict-driven backtracking-based strategy shows significant savings in the number of infeasible paths explored, by an order of magnitude. In the absence of infeasible paths in the program-under-test, the reachability-guided strategy provides significant reduction in the number of feasible paths explored and test generation time. When the program being tested has several infeasible paths, using the conflict-driven nonchronological

backtracking strategy is advantageous. The two strategies are complementary to each other and improve the overall speed of test generation.

Our proposed error-directed search strategies are aimed at catching bugs in a program faster, by targeting those parts of the program where bugs are likely to be found or that are hard to reach. The assertion-directed search strategy targets programmer-inserted assertions in the program, while achieving complete branch coverage of the program. Using this strategy, we are able to detect assertion failures in the program early in the testing process. We also propose a strategy aimed at targeting hard-to-detect errors in the program-under-test. In this work, we define these hard-to-detect errors as those that are hidden in hard to reach parts of the program. Based on this idea, we target those branches of the program that are less likely to be exercised. In this manner, our error-targeted strategies aid in biasing the order of path exploration towards parts of the program where bugs are likely to be hidden. Thus, by using a metric to target bugs in the program, in conjunction with our RG+CB strategy, we can catch errors early, while still achieving complete branch coverage quickly.

There are several opportunities for future work. Stricter metrics can be used to define hard-to-detect errors and integrated into our overall path-exploration framework. These metrics could be based on the length of the path to be explored or the complexity of the constraints in the path. We can further use a dynamically updated branch weight instead of the current proposed static *detectability* weight. For example, on encountering an infeasible path, the branch weights could be updated using information from the unsatisfiable core obtained from the SMT solver. Our search techniques have been implemented in a DFS framework but can be adapted to other path-based frameworks. It will also be interesting to combine our proposed method with other pruning heuristics.

Bibliography

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

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

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

[4] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *In 19th European Conference Object-Oriented Programming*, pp. 504–527, 2005.

[5] B. Korel, "Automated Software Test Data Generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

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

[7] P. D. Coward, "Symbolic Execution Systemsi - A Review," *Software Engineering Journal*, vol. 3, no. 6, pp. 229–239, 1988.

[8] 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, September 2008.

[9] P. Boonstoppel, C. Cadar, and D. R. Engler, "RWset: Attacking Path Explosion in Constraint-Based Test Generation," in *TACAS*, pp. 351–366, 2008.

[10] B. T. Technology, "Who is Using BullseyeCoverage," *http://www.bullseye.com/successWho.html*.

[11] B. T. Technology, "BullseyeCoverage - Measurement Technique," *http://www.bullseye.com/measurementTechnique.html*.

[12] S. Bardin and P. Herrmann, "Pruning the Search Space in Path-Based Test Generation," in *ICST '09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, (Washington, DC, USA), pp. 240–249, IEEE Computer Society, 2009.

[13] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[14] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[15] G. Nelson and D. Oppen, "Simplification by cooperating decision procedures," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 2, pp. 245–257, 1979.

[16] R. Shostak, "A practical decision procedure for arithmetic with function symbols," *Journal of the ACM (JACM)*, vol. 26, no. 2, pp. 351–360, 1979.

[17] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL (T): Fast decision procedures," in *Computer aided verification*, pp. 293–295, Springer, 2004.

[18] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. Van Rossum, and R. Sebastiani, "Efficient satisfiability modulo theories via delayed theory combination," in *Computer Aided Verification*, pp. 335–349, Springer, 2005.

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

[20] I. Johnson, "Formal Verification with SMT Solvers: Why and How," 2009.

[21] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, (London, UK), pp. 213–228, Springer-Verlag, 2002.

[22] G. Necula, "CIL Home page, 2005," *http://hal.cs.berkeley.edu/cil*, 2005.

[23] K. Kähkönen, "Evaluation of Java PathFinder Symbolic Execution Extension," 2007.

[24] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th international conference on Software Engineering*, pp. 416–426, IEEE Computer Society, 2007.

[25] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, 2008.

[26] P. Godefroid, M. Levin, D. Molnar, *et al.*, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium*, Citeseer, 2008.

[27] P. Godefroid, "Compositional Dynamic Test Generation," in *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 47–54, ACM, 2007.

[28] S. Anand, P. Godefroid, and N. Tillmann, "Demand-Driven Compositional Symbolic Execution," in *TACAS*, vol. 4963 of *LNCS*, pp. 367–381, Springer, 2008.

[29] A. Tomb, G. Brat, and W. Visser, "Variably interprocedural program analysis for runtime error detection," in *Proceedings of the 2007 international symposium on Software testing and analysis*, p. 107, ACM, 2007.

[30] R. Majumdar and R. Xu, "Directed test generation using symbolic grammars," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 134–143, ACM New York, NY, USA, 2007.

[31] R. Grosu and S. Smolka, "Monte Carlo model checking," *Lecture notes in computer science*, pp. 271–286, 2005.

[32] K. Sen, M. Viswanathan, and G. Agha, "Statistical model checking of black-box probabilistic systems," *Lecture notes in computer science*, pp. 202–215, 2004.

[33] K. Sen and G. Agha, "Automated systematic testing of open distributed programs," *Lecture notes in computer science*, pp. 339–356, 2006.

[34] M. Dwyer, S. Elbaum, S. Person, and R. Purandare, "Parallel randomized state-space search," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pp. 3–12, 2007.

[35] P. Godefroid and S. Khurshid, "Exploring very large state spaces using genetic algorithms," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6, no. 2, pp. 117–127, 2004.

[36] J. Marques-Silva and K. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[37] S. Ranise and C. Tinelli, "The SMT-LIB Standard: Version 1.2," *Department of Computer Science, The University of Iowa, Tech. Rep*, 2006.

[38] A. Cimatti, A. Griggio, and R. Sebastiani, "A simple and flexible way of computing small unsatisfiable cores in sat modulo theories," *Theory and Applications of Satisfiability Testing–SAT 2007*, pp. 334–339, 2007.

[39] I. Lynce and J. Marques-Silva, "On computing minimum unsatisfiable cores," *SAT*, vol. 4, 2004.

[40] M. Mneimneh, I. Lynce, Z. Andraus, J. Marques-Silva, and K. Sakallah, "A branch-and-bound algorithm for extracting smallest minimal unsatisfiable formulas," in *Theory and Applications of Satisfiability Testing*, pp. 467–474, Springer, 2005.

[41] L. Zhang and S. Malik, "Extracting small unsatisfiable cores from unsatisfiable boolean formula," *SAT*, vol. 3, 2003.

[42] Y. Oh, M. Mneimneh, Z. Andraus, K. Sakallah, and I. Markov, "AMUSE: A Minimally-Unsatisfiable Subformula Extractor," in *Proceedings of the 41st annual Design Automation Conference*, p. 523, ACM, 2004.

[43] C. Barrett and S. Berezin, "CVC Lite: A new implementation of the cooperating validity checker category b," in *Computer Aided Verification*, pp. 19–21, Springer, 2004.

[44] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The Mathsat 4 SMT solver," in *Computer Aided Verification*, pp. 299–303, Springer, 2008.

[45] B. Dutertre and L. De Moura, "The Yices SMT solver," *Tool paper at http://yices.csl.sri.com/tool-paper.pdf*, 2006.

[46] B. Dutertre and L. De Moura, "A fast linear-arithmetic solver for DPLL (T)," in *Computer Aided Verification*, pp. 81–94, Springer, 2006.

[47] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, p. 285, IEEE Press, 2001.

[48] D. Knuth, "Estimating the efficiency of backtrack programs," *Mathematics of Computation*, vol. 29, no. 129, pp. 121–136, 1975.

[49] G. McGraw, "Automated code review tools for security," *Computer*, vol. 41, no. 12, pp. 108–111, 2008.

[50] B. Korel, "Automated test data generation for programs with procedures," in *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 209–215, ACM New York, NY, USA, 1996.

[51] K. Kähkönen, *Automated Dynamic Test Generation for Sequential Java Programs*. PhD thesis, Helsinki University of Technology, 2008.

[52] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," *Lecture notes in computer science*, vol. 3639, p. 2, 2005.