SCIENCE CHINA
Information Sciences

• RESEARCH PAPERS •

September 2011 Vol. 54 No. 9: 1797–1812
doi: 10.1007/s11432-011-4368-7

Strategies for scalable symbolic execution-driven test generation for programs

KRISHNAMOORTHY Saparya[1]*, HSIAO Michael S.[2] &
LINGAPPAN Loganathan[1]

[1]*Intel Corporation, Folsom CA* 95630*, USA;*
[2]*Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg VA* 24061*, USA*

Received February 13, 2011; accepted July 12, 2011

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 paper, 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 present experimental evidence that these strategies can significantly reduce the search space and improve the speed of test generation for programs.

Keywords test generation, software testing, symbolic execution, path explosion, conflict analysis

Citation Krishnamoorthy S, Hsiao M S, Lingappan L. Strategies for scalable symbolic execution-driven test generation for programs. Sci China Inf Sci, 2011, 54: 1797–1812, doi: 10.1007/s11432-011-4368-7

1 Introduction

Today, testing is widely recognized as a crucial step in software development and usually accounts for about 50% of the development cost [1]. Efficient testing of a program requires the application of relevant test inputs to the program under test. Manual testing is labor-intensive and cannot guarantee that all possible behaviors of the program have been verified [2]. 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 [3]. 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 can be extremely small.

Constraint-based testing [4] 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 [5].

*Corresponding author (email: saparya.krishnamoorthy@intel.com)

Symbolic techniques have been shown to be very effective in path-based test generation; however, they fail to scale to large problems [6]. 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 [7]. However, covering all paths of a program is not the primary objective of our experiments, as is the case with most modern testing practices. As an example, we cite the tool BullseyeCoverage which is widely used [8] in the software industry to measure the quality of test data. This tool reports code coverage as function and branch coverage only [9].

In this paper we focus on tackling this path explosion problem 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 two 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. These 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 procedure.

This paper is structured as follows. Section 2 discusses some preliminaries and section 3 gives an overview of related work in this area. In section 4 and section 5 we present our search strategies along with simple examples. We discuss the implementation details and report the experiments that have been conducted in sections 6 and 7. Finally, section 8 provides a conclusion to the paper.

2 Overview

This section gives an overview of the preliminaries required to understand the work done towards this paper. We first describe how constraint-based symbolic execution systems work and then describe the motivation behind our proposed search strategies.

2.1 Constraint-based test generation

With the advent of advanced program analysis and constraint solving techniques, several tools like DART [4], CUTE [2], etc., utilize symbolic reasoning for dynamic test generation. 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 condition expression generates a symbolic constraint in terms of the symbolic inputs. To test a program using symbolic execution, the test generator first instruments the program under test. Code instrumentation provides a way to monitor these symbolic values and observe program execution.

Symbolic execution-based tools gather knowledge about the execution of a program using a directed search. Values are generated that satisfy the symbolic constraints generated along each execution path. Constraint solvers like Satisfiability Modulo Theories (SMT) solvers are used to solve these path constraints. In other words, path-oriented approaches to testing are built upon the idea of a path predicate. In [10], a path predicate is defined as follows:

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 $\models \phi_\pi$ 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 and forces the program to exercise path π. The process of symbolic execution with the help of path predicates can be best explained with an example. Table 1 shows how constraint-based execution is performed along a path of a program corresponding to the C code illustrated in Figure 1. Note that

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

Figure 1 An example of C program.

Table 1 Constraint-based execution along a path

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	$z_0 \neq x_0 + 6$

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 $0 \rightarrow 1 \rightarrow 2 \rightarrow (3, \text{true}) \rightarrow (4, \text{false})$ is computed with the help of the branch constraints of the path as $(y_1 = y_0 + 1) \wedge (z_0 = 3x_0) \wedge (x_0 \neq y_1) \wedge (z_0 \neq x_0 + 6)$. This, in turn, can be represented in terms of the input variables as: $(x_0 \neq y_0 + 1) \wedge (3x_0 \neq x_0 + 6)$. This path constraint represents all the input vectors that drive the program through the path shown in Table 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 tool attempts to sweep through all the paths of the program [4]. The interested reader is referred to [5] for more details on symbolic execution systems.

2.2 Basic DFS-based path exploration

The goal of most dynamic test generation engines 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 condition or a choice point in the program (if, switch, etc.), and each of these nodes has two possible edges corresponding to its two 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 [4].

The most common method to explore the program execution tree is the depth-first search (DFS) strategy. Algorithm 1 shows the steps involved in the basic DFS strategy. 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 condition on the stack. For conditions, 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 $-$cond.

We describe the basic depth-first search strategy in the context of the C program shown in Figure 1. 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.

Algorithm 1: Basic DFS strategy

```
Tests ← ∅
CurrNode ← initial node
Path ← CurrNode
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
    CurrNode ← previous node in Path /* backtrack */
  end if
end while
```

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 $(x_0 \neq y_0)$ 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 $(x_0 = y_0)$ obtained by negating the current path constraint. A possible solution to this path constraint is $\langle x_0 = 75, y_0 = 0\rangle$ and consequently the true branch of the first if-statement and the false branch of the second if-statement are executed. The predicate sequence $(x_0 \neq y_0 + 1) \wedge (3x_0 \neq x_0 + 6)$ is the path constraint. Following the DFS strategy, the last predicate of the current path constraint is negated, which results in $(x_0 \neq y_0 + 1) \wedge (3x_0 = x_0 + 6)$. A possible solution to this new path constraint is $\langle x_0 = 3, y_0 = 86\rangle$ and when the instrumented 'g' runs again, it uses these values for the input variables. 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.

2.3 Path explosion phenomenon

The most significant scalability challenge faced by path-based testing is in handling the exponential number of paths in the program. The path explosion problem is mainly due to nested calls, loops and conditions. Moreover, some path-based methods often use 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 code can be reached only with very long paths, which is a common situation [10].

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 conditions. As an example, for testing a 32-character 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 paper, we focus on tackling this path explosion problem in path-based testing. The aim is to provide heuristics to discard irrelevant paths as much as possible, thus reducing the number of solver calls and the overall computation time.

3 Related work

In the last five years, several symbolic execution-based techniques have been proposed that automatically generate test inputs. Consequently, many lines of work address the problem of path explosion, although in different ways from ours. A technique described in [6] builds on the basic depth-first search strategy to bound the number of branches explored. Burnim and Sen also describe other strategies like the uniform random search and control-flow directed search. In this CFG-directed search, the algorithm finds short paths through the static graph from branches along the execution to branches that have not yet been explored.

In [11], 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 [12], 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 [13] 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 [7] deals with pruning a path exploration when the current state is considered similar to a previously encountered state. Some work has been conducted on modular test generation in order to avoid function inlining. Some of these approaches are based on function summaries [14] while others handle functions lazily [15]. In these ways, the problem of path explosion in dynamic test generation were moderately addressed, although different from ours.

4 A reachability-guided search strategy

Our first search strategy, 'reachability-guided search' performs a reachability analysis of the control-flow graph (CFG) of the program to decide whether the current branch must be expanded 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 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 [6]. Our reachability-guided strategy utilizes the reachability graph of the program-under-test to explore important parts of the code. 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 can guide the search towards the important parts of the code, where these important parts are user-specified in terms of program conditions (or nodes in the CFG).

This technique builds on the depth-first strategy, such that this additional reachability check is performed while backtracking through the CFG. Using this new strategy, whenever the test generator encounters a program condition while backtracking, it first looks up the counter-edge of the previously visited edge. This counter-edge is explored only if it can lead to any important condition that has not yet been visited, or if the current condition 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 antecedent node in the CFG and this process continues until the root node is reached. This method helps avoid re-visiting branches that do not lead to any new, important branches. The steps of the reachability-guided strategy are described in Algorithm 2.

For programs with several interdependent function calls, the reachability-guided strategy can quickly achieve complete branch coverage. Let us consider a densely-connected control-flow graph as depicted in Figure 2, with the number of feasible paths exponential to the number of nodes in the graph. For such a program, our reachability-guided strategy would only try to explore every edge in the graph and not every feasible path. The total number of possible paths in the graph is in the order of 100, whereas the number of edges is only 20. When we are not interested in exercising every feasible path in the graph and are only interested in as many test cases to cover all edges in the graph, the reachability-guided search strategy helps us achieve exactly that.

Algorithm 2: Reachability-guided strategy

```
Tests ← ∅
CurrNode ← initial node
Path ← CurrNode
while Path is not empty do
   not covered any branches of CurrNode then
    transition ← false branch of CurrNode
    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
          record Path as infeasible
        end if
      end if
  else if covered only one branch of CurrNode then
    /* symmetric case for true branch */
  else
    CurrNode ← previous node in Path /* backtrack */
  end if
end while
```

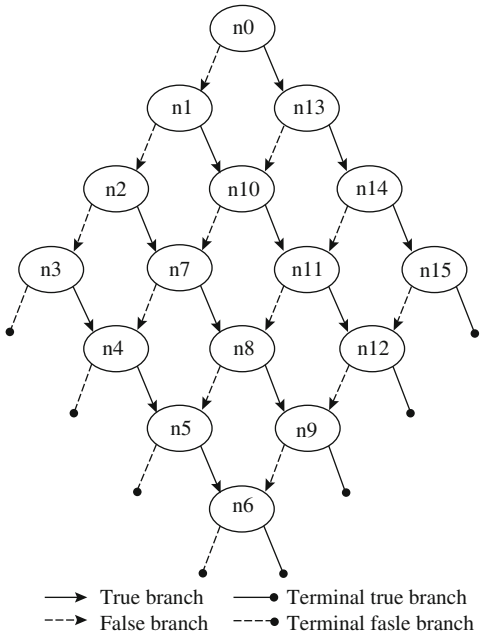

Figure 2 A control-flow graph showing how complete branch coverage can be quickly obtained using the reachability-guided strategy .

The reachability-guided search strategy provides significant improvement in speed of test case generation by achieving complete branch coverage quickly, while exploring fewer paths. In addition, by supplying a list of important conditions, we can 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 conditions 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 condition and the search terminates once the desired branch is exercised. Using the reachability-guided strategy, there is a significant reduction in the number of redundant test cases generated and infeasible paths explored.

5 A conflict-driven backtracking strategy

In the previous section, we presented an approach to ensure that each 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 programs, a 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, which is similar to the search strategy found in most of the SAT solvers today [16] and is also known as backjumping. We have extended this concept towards the problem of software test generation using SMT solvers. In addition to conflict-driven non-chronological 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 5.2.

A search engine that backtracks chronologically may 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.

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. 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. 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 multiple levels in the search tree at once.

An example of the benefit of this strategy can be seen in Figure 3. The nodes in the graph depict conditions in a program and the solid and dotted lines represent their true and false branches, respectively. The conditions 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 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 3. 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:

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

Note that four infeasible paths are explored before we backtrack to node 5 and explore the next feasible path, say $(0,F) \to (1,T) \to (3,T) \to (5,T) \to (7,F) \to (9,T)$. 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., condition $(y < 7)$. Here, for the current path $(0,F) \to (1,T) \to (3,T) \to (5,F) \to (6,T) \to (9,F) \to (10,T)$, the current decision level (at node 10) is 6 and the level that we must backtrack to, β, is 3. We then negate this condition 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!

5.1 Using the unsatisfiable core for conflict analysis

In order to implement nonchronological backtracking, it is first necessary to determine 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 instance. The SMT-LIB standard [17] describes the unsat core as a subset of the set of all assertions that the solver has determined to be unsatisfiable.

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.

Some of the current state-of-the-art SMT solvers provide ways to generate the unsat core with techniques adapted from SAT. CVC LITE [18] and a recent extension of MATHSAT [19] can compute unsatisfiable cores as a byproduct of the generation of proofs. The solver that we use, Yices [20] 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 [21]. For this purpose, each clause is represented as an assertion and each assertion in the core is identified by an ID. The unsatisfiable core is then a 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 IDs of the conflict clauses are then mapped onto the conditions 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 condition 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 3, the unsat core that we obtain from Yices contains this pair of conflicting clauses $[(y < 4)T, (y < 7)F]$. We then backjump to the previous condition that is a part of the conflict clause pair, i.e. condition $(y < 7)$. 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 condition 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 3.

5.2 Conflict-driven learning

Another pruning technique used in most state-of-the-art SAT solvers is called learning. Learning extracts and memorizes information from the previously searched space to prune the search in future. Learning

Algorithm 3: Conflict-driven backtracking strategy

```
Tests ← ∅
CurrNode ← initial node
Path ← CurrNode
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
      if CurrNode ← node in blevel /* backjump */
    else
      CurrNode ← previous node in Path /* backtrack */
    end if
  end if
end while
```

is achieved by adding clauses to the existing clause database. Here, we focus on learning that occurs as a consequence of conflicts encountered during the search process. This is referred to as conflict-driven learning and from this point on, we will use the term learning only in this context.

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 conditions 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 condition along with its direction assignment (i.e., true or false). Consequently a conflict

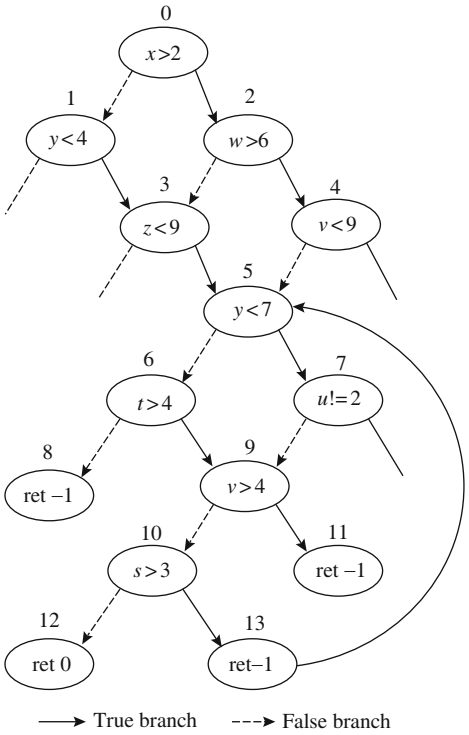

Figure 3 Control-flow graph showing the benefit of the conflict-driven backtracking strategy.

clause represents a subset of the branch assignments that led to an infeasible path. In our example depicted in Figure 3, the conflict clause learned is $[(1, T), (5, F)]$. 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.

5.3 Completeness of the algorithm

Further, we prove that our non-chronological 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 [16], since our non-chronological backtracking strategy is based on this algorithm. 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 conditions 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 condition 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 non-chronological backtracking search algorithm is complete.

Proof. It is well known that the depth-first strategy, with chronological backtracking is a complete search procedure. 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 do not 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 $\{A_\gamma \mid \beta < \gamma \leqslant \alpha\}$ 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.

6 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 [4] and CUTE [2] 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 [22], 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 under test. 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 [20].

For experiments to tackle the path explosion problem, we are mainly interested in the path exploration framework of the test generation tool. We implemented our search strategies on a stand-alone path exploration framework without the overhead of program instrumentation and symbolic execution. The proposed technique 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. It 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.

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 conditions in the program can be expressed using the available SMT theories. Programs written in any 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. Our search framework can be integrated with current constraint-based testing tools.

7 Experiments and results

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

Table 2 Experimental results for the path-exhaustive, reachability-guided and conflict-driven backtracking strategies[a)]

		Path-exhaustive (PE)			Reachability-guided (RG)			Conflict-driven backtrack (RG+CB)			
Test-case	# conds	Paths exp.	Inf. paths	Time (s)	Paths exp.	Inf. paths	Time (s)	Paths exp.	Inf. paths	Time (s)	Speedup
tc1	17	6816	576	5.58	32	12	0.14	19	1	0.07	2.0
tc2	20	12944	33136	23.48	1640	5574	5.68	18	9	0.06	94.67
tc3	28	3238240	73760	2596.01	28	137	0.53	28	3	0.16	3.31
tc5	30	4239360	0	2701.85	31	0	0.12	31	0	0.12	1.0
tc18	34	–	–	–	407	192	4.14	34	1	0.28	14.79
tc6	39	–	–	–	9590	9568	73.67	41	1	0.28	263.11
tc19	39	–	–	–	35	843264	525.41	35	2	0.18	2918.94
tc7	42	–	–	–	–	–	–	42	4	0.25	N/A
tc20	42	1716	0	0.93	43	0	0.06	43	0	0.06	1.0
tc8	47	–	–	–	3868	3840	55.91	53	4	0.53	105.49
tc9	61	–	–	–	127	50	2.59	62	3	0.71	3.65
tc22	84	1159488	0	1026.91	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

a) '–' indicates time-out in 2 h.

promising and show that the non-chronological 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 2.8 GHz Intel® Core 2 Duo™ machine with 2.9 GB of RAM.

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 paper 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.

We compare our two 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 non-chronological backtracking strategy performs conflict analysis to determine the level to backtrack to and uses conflict-driven learning to avoid revisiting already explored infeasible paths. We refer to this combined strategy as the conflict-driven backtracking strategy. We also 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 [6].

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 2. In this table, for each test case (which is derived from a Test Program), # conds denotes the number of distinct conditions in the program model, where each condition leads to a true and false transition. Based on the outcome

Table 3 Statistics of running experiments with reachability-guided and conflict-driven strategies, for varying number of conflicts[a)]

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	1467	1.16	20	4	12	0.06
		8	1640	5574	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	2119	2654	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	2008	9115	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

a) '–' indicates that no back-jumping was performed.

of the condition, each transition leads to either another such state or a terminal state in the model. In Table 2, 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 h). In each of the experiments, statistics are shown for complete branch coverage of the model-under-test.

We can see that the RG strategy greatly outperforms the PE strategy for all cases. For most test cases with more than 30 distinct conditions (except test cases without any infeasible paths), the PE strategy results in a time-out. We also observe that, unsurprisingly, the RG and RG+CB strategies give the same results for test cases 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 model size and the number of infeasible paths increase, 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 test cases, 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 conditions, the path-exhaustive method timed out in 2 h. The reachability-guided method needed to explore 407 paths to complete the search. The total time taken is less than 5 s! 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 3 shows the results for varying the number of conflicts introduced into the model. For each test case derived from a Test Program, # 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 model. # conds, Paths exp., Inf. paths and Time (s) retain the same meanings as in Table 2.

We can see from Table 3 that, when there are no inconsistencies in the model, the two strategies 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 paths explored under the RG strategy markedly increases and consequently, so

Table 4 Results for comparison of conflict-driven backtracking search strategy with two CREST search strategies[a)]

		CREST (UR)		CREST (CFG-D)		RG+CB		Speedup of RG+CB	
Test-case	# conds	# iters	Time (s)	# iters	Time (s)	# iters	Time (s)	vs. (UR)	vs. (CFG-D)
tc2	20	74	1.77	19	0.39	20	0.06	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	4779	32.64	**116**	2.24	**85**	0.21	153.95	**10.56**
tc23	143	1058	29.29	159	3.77	194	5.52	5.31	0.68
tc24	210	53524	393.76	293	5.87	211	0.83	475.33	7.08

a) The values in boldface are used to highlight test cases with large speed-up.

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 2. In the case of the RG+CB strategy, with the help of the intelligence of the conflict analysis engine, infeasible paths once explored, are not visited again. For example, for tc2 with 20 conditionals, both the RG and the RG+CB method achieved complete branch coverage in less than 0.1 s, 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 s 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 s, more than 90× speedup over the RG method.

In Table 4, we compare our conflict-driven backtracking-based (RG+CB) search strategy with two search strategies in CREST [6]. 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 4 as CREST (UR) and CREST (CFG-D) respectively. For each test case, # conds represents the number of conditions 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 test case 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 4, 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 conditions, the CREST (UR) method required 196 iterations to achieve complete branch coverage and the CREST (CFG-D) method required 176, whereas our method required just 35. 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. 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.

8 Conclusions

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 paper 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. Our experiments showed that the reachability-guided strategy can reduce the number of tests required to achieve branch coverage quickly. The nonchronological backtracking strategy shows significant savings in the number of infeasible paths explored, by an order of magnitude. The two strategies are complementary to each other and improve the overall speed of test generation. There are several opportunities for future work. These techniques have been implemented in a DFS framework but can be adapted to other path-based frameworks.

Acknowledgements

This work was supported in part by a grant from Intel® Corporation.

References

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

2 Sen K. Marinov D, Agha G. CUTE: A concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, 2005. 263–272

3 Pacheco C, Ernst M D. Eclat: Automatic generation and classification of test inputs. In: Proceedings of the 19th European Conference Object-Oriented Programming, Glasgow, UK, 2005. 504–527

4 Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design And Implementation, New York, NY, USA, 2005. 213–223

5 Coward P D. Symbolic execution systems—a review. Softw Eng J, 1988, 3: 229–239

6 Burnim J, Sen K. Heuristics for scalable dynamic test generation. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, Italy, 2008. 443–446

7 Boonstoppel P, Cadar C, Engler D R. RWset: Attacking path explosion in constraint-based test generation. In: Proceedings of the 14th International Conference on Tools and Algorithms for Construction and Analysis of Systems, Budapest, Hungary, 2008. 351–366

8 Technology B T. Who is Using BullseyeCoverage. http://www.bullseye.com/successWho.html

9 Technology B T. BullseyeCoverage-Measurement Technique. http://www.bullseye.com/measurementTechnique.html

10 Bardin S, Herrmann P. Pruning the search space in path-based test generation. In: Proceedings of the 2009 International Conference on Software Testing Verification and Validation, Washington DC, USA, 2009. 240–249

11 Majumdar R, Sen K. Hybrid concolic testing. In: Proceedings of the 29th International Conference on Software Engineering, Minneapolis, MN, USA, 2007. 416–426

12 Cadar C, Ganesh V, Pawlowski P, et al. EXE: Automatically generating inputs of death. ACM Trans Inf Syst Secur, 2008, 12: 10

13 Godefroid P, Levin M, Molnar D, et al. Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 2008

14 Godefroid P. Compositional dynamic test generation. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, 2007. 47–54

15 Anand S, Godefroid P, Tillmann N. Demand-Driven Compositional Symbolic Execution. Technical Report MSR-TR-

2007-138. 2008

16 Marques-Silva J, Sakallah K. GRASP: A search algorithm for propositional satisfiability. IEEE Trans Comput, 1999, 48: 506–521

17 Ranise S, Tinelli C. The SMT-LIB Standard: Version 1.2. Technical Report, Department of Computer Science, The University of Iowa. 2006

18 Barrett C, Berezin S. CVC Lite: A new implementation of the cooperating validity checker category b. In: Computer Aided Verification. New York: Springer, 2004. 19–21

19 Bruttomesso R, Cimatti A, Franzén A, et al. The Mathsat 4 SMT solver. In: Computer Aided Verification. New York: Springer, 2008. 299–303

20 Dutertre B, De Moura L. The Yices SMT Solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, 2006

21 Dutertre B, De Moura L. A fast linear-arithmetic solver for DPLL (T). In: Computer Aided Verification. New York: Springer, 2006. 81–94

22 Necula G C, McPeak S, Rahul S P, et al. CIL: Intermediate language and tools for analysis and transformation of C programs. In: Proceedings of the 11th International Conference on Compiler Construction, London, UK, 2002. 213–228