ATPG based Preimage Computation: Efficient Search Space Pruning using ZBDD

Kameshwar Chandrasekar

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

Master of Science
in
Electrical Engineering

Dr. Michael S. Hsiao, Chair
Dr. James R. Armstrong
Dr. Dong S. Ha

28 July, 2003
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: Preimage, ATPG, ZBDD, Equivalence Checking, Model Checking

ATPG-based Preimage Computation: Efficient Search Space Pruning using ZBDD

Kameshwar Chandrasekar

Abstract

Preimage Computation is a fundamental step in Formal Verification of VLSI designs. Conventional OBDD-based methods for Formal Verification suffer from spatial explosion, since large designs can blow up in terms of memory. On the other hand, SAT/ATPG based methods are less demanding on memory. But the run-time can be huge for these methods, since they must explore an exponential search space. In order to reduce this temporal explosion of SAT/ATPG based methods, efficient learning techniques are needed.

Conventional ATPG aims at computing a single solution for its objective. In preimage computation, we must enumerate all solutions for the target state during the search. Similar sub-problems often occur during preimage computation that can be identified by the internal state of the circuit. Therefore, it is highly desirable to learn from these search-states and avoid repeated search of identical solution/conflict subspaces, for better performance.

In this thesis, we present a new ZBDD based method to compactly store and efficiently search previously explored search-states. We learn from these search-states and avoid repeating subsets and supersets of previously encountered search spaces. Both solution and conflict subspaces are pruned based on simple set operations using ZBDDs. We integrate our techniques into a PODEM based ATPG engine and demonstrate their efficiency on ISCAS '89 benchmark circuits. Experimental results show that upto 90% of the search-space is pruned due to the proposed techniques and we are able to compute preimages for target states where a state-of-the-art technique fails.

To my Parents and Brother

without whom nothing would have been possible.

Acknowledgements

It is a pleasure to acknowledge all the people who made this work possible. I would like to express my sincere thanks to my advisor Dr. Michael S. Hsiao, for his inspiration and support throughout my graduate program. I would like to thank Dr. James R. Armstrong and Dr. Dong S. Ha for serving on my thesis committee. Last but not the least, I thank my friends and relatives for their emotional support and encouragement.

Kameshwar Chandrasekar

July 2003

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Growing advances in VLSI technology have lead to an increased complexity in hardware systems. It is imperative to verify the correctness of these systems right at the design stage. Bugs in a design that are not discovered in early stages can be extremely expensive to correct later. The complexity of large designs poses serious challenges to the verification community.

1.1 Design Verification

Design verification is the process of checking if a design implementation conforms to its specifications of functionality, timing, testability and power dissipation. In [1], functional verification methods are classified into three categories:

1. Software Simulation
2. Hardware Emulation
3. Formal Verification

Software Simulation has traditionally been used to verify the correct operation of designs. Since it is not practically feasible to simulate all possible input patterns, an intelligent subset of test patterns is simulated. Although simulation based methods account for most of the design errors, they may potentially miss some of the corner-case errors. They are incomplete since all the test patterns are not simulated.

Hardware Emulation uses Field Programmable Gate Array (FPGA) chips to speed up the verification process by several orders of magnitude. However, hardware emulators are expensive and it is time-consuming to map the design to an emulator.

Recently, *Formal Methods* are gaining importance for verifying hardware correctness. Formal Methods refer to the application of mathematical reasoning for verification of VLSI circuits that are considered as Finite State Machines (FSM). Correctness of hardware systems that are formally verified hold for all input patterns. A wide variety of formal methods have been proposed for design verification. Two methods that have recieved significant attention are *Equivalence Checking* and *Model Checking*.

- *Equivalence Checking*: This method aims at checking the equivalence of two designs at the same or different levels of abstraction, in the design process. To verify the equivalence of two sequential circuits, a *miter circuit* of the specification and implementation is formed as shown in Figure 1.1. The corresponding input pairs of the two circuits are connected together and the output pairs are fed into an XOR gate. The equivalence of the two circuits can be asserted if the primary output response of the *miter circuit* is tautology '0' for any input vector and *reachable state*. In order to compute the set of *reachable states*, Finite State Machine Traversal is performed prior to equivalence checking.

- *Model Checking*: Model Checking verifies if a circuit satisfies certain properties or not. The design specification is modeled as a temporal logical formula, while the

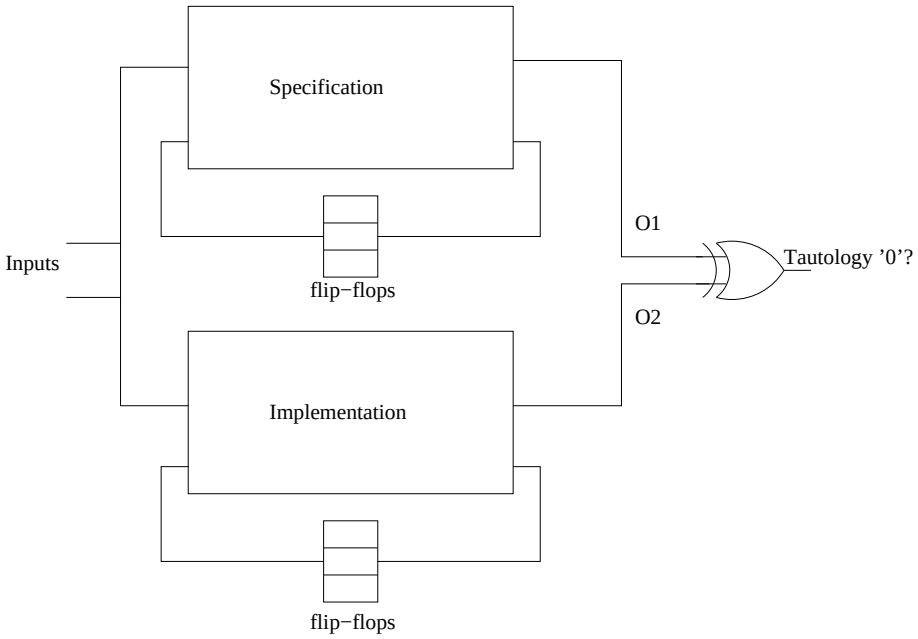


Figure 1.1: Miter Circuit for Equivalence Checking

implementation is described as a Finite State Machine. Model Checking requires to verify if a specific property holds in the different states of circuit. This in turn requires FSM traversal to compute the set of reachable states.

1.2 FSM Traversal

Formal Verification relies on FSM Traversal for both Equivalence Checking and Model Checking. FSM Traversal aims at computing the set of reachable states, starting from a given initial state for the Finite State Machine. In earlier methods for Formal Verification, the FSM was explicitly represented as a State Transition Graph (STG). Given a State Transition Graph for an FSM, the complete set of reachable states can be computed by traversing the STG in a breadth first or depth first manner. For the STG shown in Figure 1.2, FSM traversal

Table 1.1: Fixed Point Iteration for Reachable states of FSM Traversal

Iteration	States Reached
0.	{S0}
1.	{S0, S1, S2}
2.	{S0, S1, S2, S3, S4, S5}
3.	{S0, S1, S2, S3, S4, S5, S6}
4.	{S0, S1, S2, S3, S4, S5, S6}

is performed explicitly as shown in Table 1.1. For large number of states, it is difficult to manage an STG and *explicit* traversal becomes time-consuming.

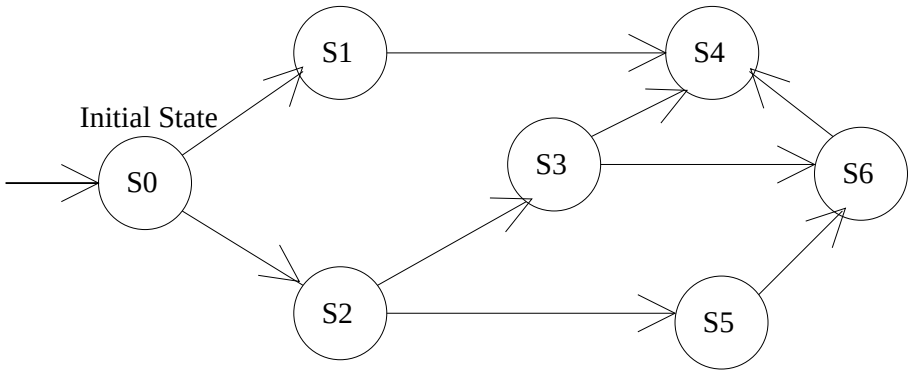


Figure 1.2: Explicit Method for State Space Traversal

Symbolic methods are proposed as an alternative to explicit methods. The FSM and the states are represented by logical formulas in these methods. The Transition Relation (T), present states (X) and next states (Y) that represent the FSM are graphically shown in Figure 1.3. The image/preimage for a set of states are computed by Existential Quantification of the Transition Relation, with respect to the state variables.

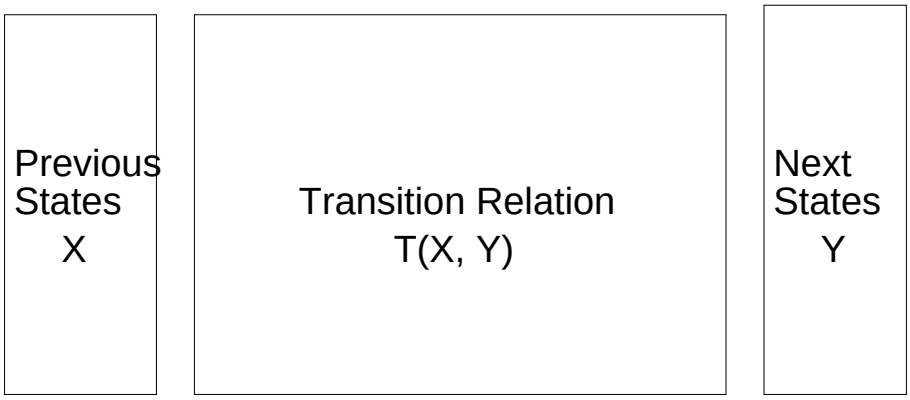


Figure 1.3: Symbolic Method for State Space Traversal

1.2.1 Image Computation

Given an initial state I(X) and a Transition Relation (T) for an FSM, computation of the set of next states is called image computation. Image computation is performed iteratively till a fixed point is reached to compute all the reachable states. The complete image set for FSM traversal, shown in Table 1.1, is computed using symbolic methods as shown in Figure 1.4. The image set computed for one-iteration becomes present states for the next iteration. In this way, the next-states are computed iteratively to obtain the complete set of reachable states.

Mathematically,

$$Image(Y) = \exists_x T(X, Y).I(X)$$

where,

- X represents initial state elements
- Y represents next state elements

- I(X) is the set of initial states
- T(X,Y) is the Transition Relation

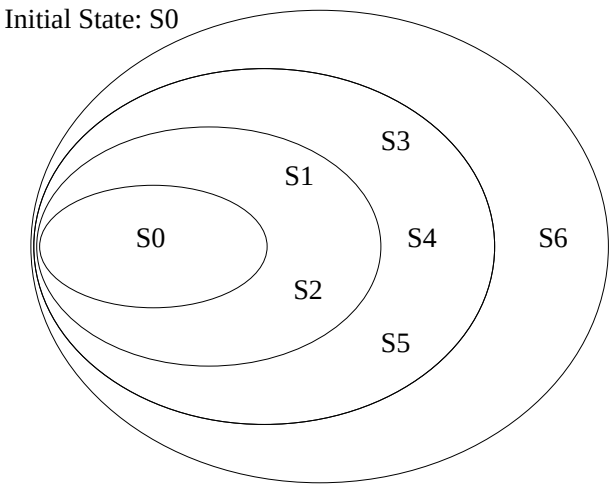


Figure 1.4: FSM Traversal - Image Computation

For *Equivalence Checking*, it is verified that the miter output is always '0' for all combinations of input vectors starting from the given initial state. For *Model Checking*, it is verified if the given property is satisified in the reachable states, depending on the nature of the property.

1.2.2 Preimage Computation

Preimage Computation is the problem of finding the set of all previous states (X) that can reach a given set of target states (Y). In a sequential circuit, the flip flops are the state elements and the circuit represents the Transition Relation (T). The graphical representation of the problem is shown in Figure 1.5. Considering $S6$ as the target state for the STG in Figure 1.2, preimage computation is performed iteratively till a fixed point is reached.

The mathematical formulation of the problem is,

$$Preimage(X) = \exists_y T(X,Y).I(Y)$$

where,

- X represents previous state elements
- Y represents next state elements
- I(Y) is the set of target states
- T(X,Y) is the Transition Relation

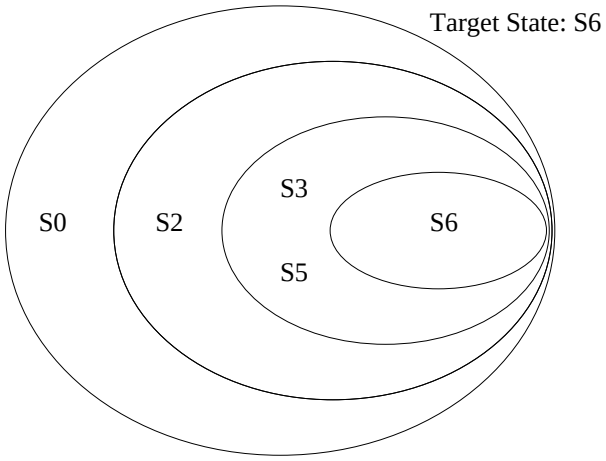


Figure 1.5: FSM Traversal - Preimage Computation

In *Equivalence Checking*, the set of preimages are computed with the output of the miter as '1'. If no legal states are found in the preimage, then the two circuits are asserted to be equivalent. For *Model Checking*, a bad property is asserted as the target state. If no legal states are found, then the property is satisfied.

In this work, we focus on preimage computation for a gate-level abstraction of the circuit.

1.3 Previous Work

Symbolic methods based on Reduced Ordered Binary Decision Diagrams (ROBDD) have widely been used for image and preimage computation. In these methods, the Transition Relation and the set of states are represented as an ROBDD. Existential Quantification is performed as a Boolean operation and the preimage is coherently represented as another ROBDD. However, ROBDD cannot be constructed for Transition Relation of larger circuits due to the memory explosion problem. These methods are suitable only for small and medium sized circuits. In [2, 3], partitioned ROBDDs were introduced to represent the Transition Relation for larger circuits. However, these methods also suffer from peak-memory explosion problem during existential quantification.

To avoid the memory explosion problem of ROBDDs, alternate techniques were explored. A Reduced Boolean circuit (RBC) to represent the Transition Relation is proposed in [4] and certain rules for efficient variable quantification are provided for the RBC. However, the quantification rules are applicable only for specific formula structures. In the worst case, they do quantification $\exists_x f(X)$ by generating two formulas, $f_{\overline{x}}$ and f_x and taking the disjunction of the two. The length of these formulas can increase exponentially as well. In [5], the Transition Relation is represented as a Boolean Expression Diagram (BED). The quantification procedure in their work is also specific to the structure of formulas.

Recently, SAT/ATPG based methods are gaining importance in formal verification. Compared to OBDD, SAT/ATPG trades off time with space. In SAT based methods, the Transition Relation is represented in Conjunctive Normal Form (CNF). Existential Quantification is performed by invoking a SAT solver and finding all the satisfying assignments for the CNF. SAT methods were integrated with ROBDDs in [6]. The Transition Relation is representated as a CNF and the set of states are represented by BDD. A SAT solver performs

high-level decomposition of the search-space and BDD is used to compute all solutions below intermediate points in SAT decision tree. This method still falls into the framework of partitioned BDDs. McMillan showed that pure SAT based methods are suitable for Unbounded Model Checking, in [7]. The efficiency of these methods lies in *learning* from the satisfying assignments and conflicts encountered during the search. On the other hand, structural information added to pure SAT based methods showed significant results for image computation in [8]. The structure of the circuit guides the decision engine of SAT to make intelligent decisions and avoids unnecessary variable-assignments.

In ATPG based methods, the Transition Relation is represented by a levelized circuit. Existential Quantification is performed by finding all solutions that can justify the target state. The set of all solutions forms the complete preimage, at the end. In [9], the authors show that ATPG based techniques outperform SAT, when considering very large designs over multiple timeframes, for Bounded Model Checking. The structural information is already available to an ATPG engine and the decisions are made, based on a simple backtrace [10] algorithm. In SAT/ATPG based techniques, *learning* plays a very important role to prune the search-space. The knowledge can be in the form of implications [11], assertions [12] or conflict clauses [13, 14, 15]. In [16], the authors show that SAT based methods are more suitable for conflict analysis and conventional conflict-driven learning in ATPG is complicated. Efficient learning techniques and manipulation of knowledge, is required to reduce the overhead in storing and using the knowledge base.

In [17], 'Success-driven learning' efficiently prunes the search-space for 'ATPG based preimage computation', based on identical solution-subspaces. After each decision, an internal cut-set of the circuit is identified as a search-state. Equivalent search-states that lead to the same solution subspace are identified to avoid exploring the same search-space again. The Decision Tree obtained during solution-search is stored as a BDD that represents the complete preimage set. Since solution subspaces heavily overlap during preimage computation,

considerable savings is obtained in terms of time and memory.

1.4 Contribution

In this work, we explore *ATPG based preimage computation* and present efficient learning techniques to prune the search-space. The Transition Relation is represented as a levelized sequential circuit. Existential Quantification is performed by invoking a PODEM-based ATPG algorithm to find all solutions that justify the target state. The Decision Tree obtained during the search is stored as a Free BDD to represent the complete preimage set at the end. We present learning techniques, based on the state of the circuit after each decision, to facilitate faster Existential Quantification and sharing of subspaces in the Decision Tree.

The contribution of this thesis is three-fold:

1. We identify partially-equivalent search-states as soon as possible during the search process and assign necessary additional input values to quickly reach the equivalent search-state.

2. As a dual of pruning solution subspaces, we employ *search-state based conflict driven learning* to prune the conflict subspaces of the Decision Tree.

3. A *Novel ZBDD based method* is proposed for compact storage of search-states and to manipulate them, for efficient search-space pruning.

1.5 Thesis Organization

The rest of the thesis is organized as follows.

- Chapter 2 introduces the preliminary concepts and the terms used in the sequel.
- Chapter 3 describes the concept of *'Augmented Success Driven Learning'* that is used to learn from previous solutions obtained during ATPG.
- Chapter 4 explains the concept of *'Search State based Conflict Driven Learning'* that is used to learn from conflict subspaces of the search-space.
- Chapter 5 integrates success-driven and conflict-driven approaches and shows significant search-space pruning for preimage computation.
- Chapter 6 concludes the thesis with a few recommendations for future work.

Chapter 2

Background

Symbolic methods to solve the problem of preimage computation include Reduced Ordered Binary Decision Diagrams (ROBDD), Satisfiability (SAT) and Automatic Test Pattern Generation (ATPG). We focus on ATPG based preimage computation. For a detailed introduction to ROBBD and SAT based methods, the reader is referred to [18, 19, 20, 7, 21]. In this Chapter, we present the background pertaining to ATPG and our work. The basics of *ATPG based preimage computation* are explained. We describe *Success Driven Learning* for efficient preimage computation [17]. The technical terms used to explain the rest of the thesis are introduced. Finally, a brief overview on Zero-Suppressed Binary Decision Diagrams (ZBDD) is presented.

2.1 ATPG based preimage computation

An ATPG algorithm implicitly explores the entire search-space to generate a solution. For preimage computation, after a solution is found we must backtrack and search for the next solution. Each decision is considered a node in the Decision Tree and the entire search-

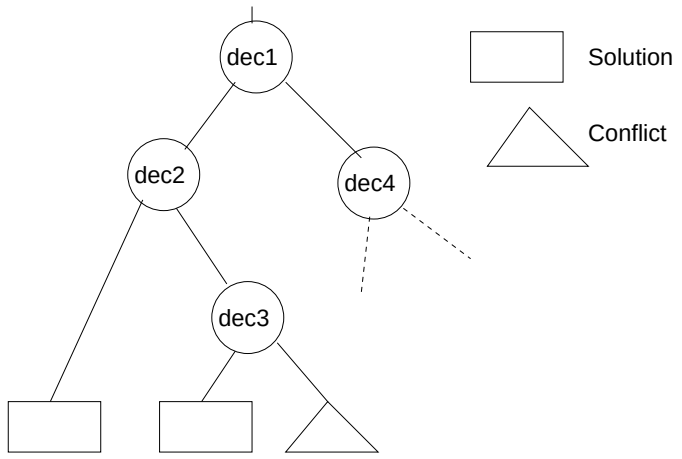
space is explored incrementally, to find all solutions. The solution set forms the complete preimage at the end. Efficient *learning* techniques help to prune the search-space and improve the performance of the ATPG engine. The prototype ATPG algorithm used in this work is PODEM [10]. A naive PODEM algorithm to compute preimage is shown in Figure 2.1(A).

```
function preimage() {
 // step 1. Terminal Conditions
    if (objective_satisfied()) {
       update_solution_BDD();
       return;
    }
    if (objective_violated()) return;

 // step 2. get new decision node
    decision_node = backtrace();
    imply();
    preimage();

 // step 3. explore other branch
    flip_decision(decision_node);
    imply();
    preimage();
}
```

(A) Algorithm



(B) Decision Tree

Figure 2.1: Naive Algorithm for Preimage Computation

The target state is the ATPG objective. Decisions are chosen based on a simple backtrace from the objective. The logic implications of each decision are deduced. If the objective is satisfied, the input assignments form a solution. The solution is stored and then we proceed to find the next solution. If the objective is violated, the input assignments form a conflict. We flip the most recent decision and search for a solution. Otherwise, we decide on another input and continue the search, until the entire search-space is explored. The corresponding Decision Tree is shown in Figure 2.1(B). The entire search-space under the tree is explored to find all the solutions. Conventionally, ATPG performs fault justification and fault propagation. However, in preimage computation it is sufficient to justify the target state and we are not concerned with fault propagation phase. Unlike conventional ATPG,

we backtrack even after a solution is found, as shown in Step 2. In this way, all solutions are found, which form the complete preimage set at the end. An example is shown in Figure 2.2.

The objective of the ATPG engine is to justify $z = 1$. We backtrace through gate g and make a decision $d = 1$. The logic implications of $d = 1$, is recorded. Since the objective is neither satisfied nor violated we continue the search. The search process is tabulated in the Table in Figure 2.2(B).

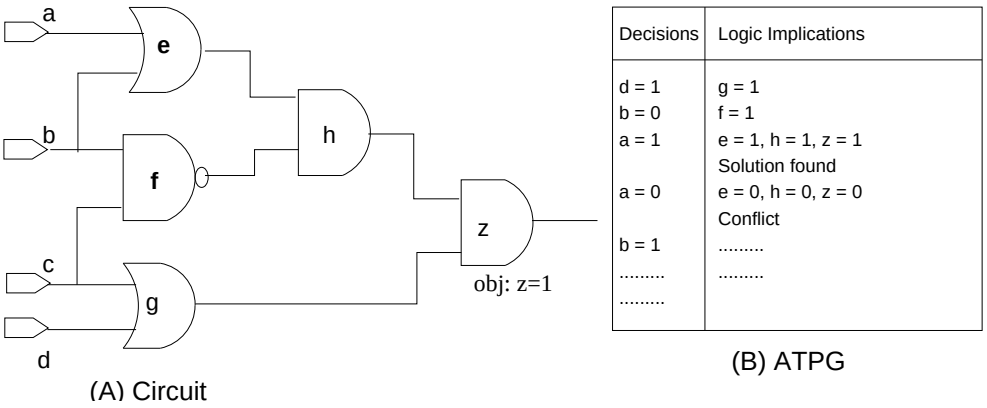


Figure 2.2: PODEM based ATPG

2.2 Success Driven Learning

Efficient *learning techniques* are required to speed up ATPG, since it is inherently limited in time. Conventional *learning* techniques include implications [11], dependency directed backtracking [22] and conflict analysis [15]. Certain learning techniques for ATPG like X-Path check and Unique Point Sensitization are not suitable for preimage computation, since they are incorporated in the fault propagation phase. Recently, *Success Driven Learning* has been proposed for 'ATPG based preimage computation', in [17]. A few terms are introduced,

before explaining the concept of *Success Driven Learning*.

1. **Decision Tree:** The Tree obtained by the branch-and-bound procedure of ATPG, with input assignments as internal decision nodes, is called the Decision Tree. Each node has two branches. The left and right branches of each decision node represent 1 and 0 input-assignment respectively. Two terminal nodes, TERMINAL-1 and TERMINAL-0, represent solution assignment and conflict assignment for the corresponding path in the Decision Tree.

2. **Search-State:** After each decision, logic simulation of the partial input assignment forms a decomposition in the circuit. The internal state of the circuit after each input assignment is considered a *search-state* in the Decision Tree.

3. **Cut-set for search-state:** Each search-state can be uniquely represented by a cut-set of the circuit. A simple backtrace from the ATPG objective can identify this unique internal circuit state. Essentially, the first frontier of specified nodes, encountered during backtrace, is the search-state representative.

   For a circuit in Figure 2.3(A), a portion of the Decision Tree is shown in Figure 2.3(B). The sets along the branches (eg. $\{g\}$ and $\{g, f, \bar{b}\}$) are the cut-sets for the corresponding search states. For the last branch (b=0), the cut-set $\{g, f, \bar{b}\}$ is denoted by the dotted line in the circuit.

4. **solution/conflict branch:** A branch in the Decision Tree that has *at least-one/no* solution below it.

5. **solution/conflict cut-set:** A cut-set for the search-state in the *solution/conflict* branch.

6. **solution/conflict subspace:** A search subspace below a *solution/conflict* branch.

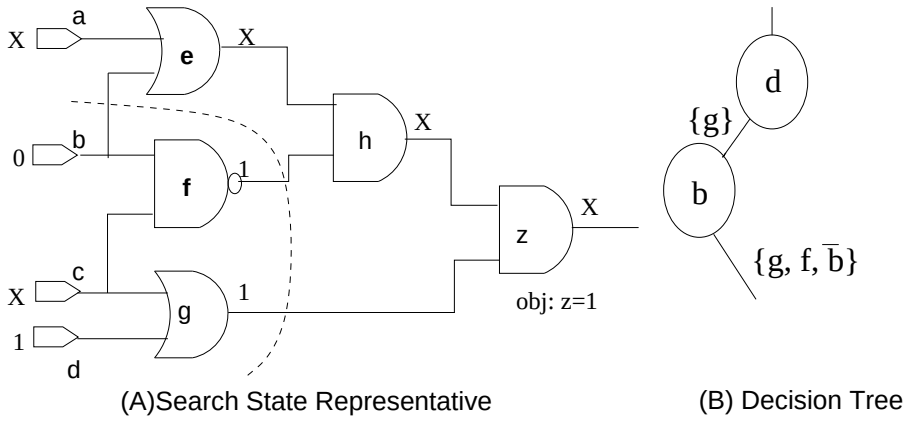


Figure 2.3: Preliminaries

Solution subspaces heavily overlap during preimage computation. In order to avoid searching the same solution subspaces again, *Success Driven Learning* has been introduced. After each decision, a cut-set for the current branch in the Decision Tree is identified. If a solution is found below that branch, the corresponding cut-set and the link to branch in Decision Tree is recorded in a Hash-table. In future, after every decision, the current cut-set is searched in the Hash-table for equivalence. If we detect a Hash-hit, the corresponding link in the Decision Tree is connected to the current branch. The Decision Tree forms a BDD that becomes a complete representation of the preimage, at the end. This technique *learns* from solution subspaces and prunes the search-space. An example for *Success Driven Learning* is illustrated in Chapter 3, before introducing the idea of *Augmented Success Driven Learning*.

2.3 Set manipulation using ZBDD

In [23], Minato introduced ZBDDs to efficiently represent *sets of combinations*. ZBDDs overcome the spatial explosion problem of ROBDDs and are known for their compact representation of sparse sets. In [23], efficient algorithms were also presented to perform set

operations. Originally, they were used to solve unate covering problems, graph optimization problems and logic minimization problems. Recently, ZBDD based methods are gaining importance in SAT-solvers [24, 25] and path delay computation [26] also.

Sets of combinations S = {{a,b}, {a,c}, {c}} are represented in a ZBDD as shown in Figure 2.4(A). Each path from the root to TERMINAL-1 represents a set in the ZBDD. The left branch represents the 1-edge and the right branch represents the 0-edge for every node. A 1-edge from a node denotes the presence of the element in the set and a 0-edge denotes its absence in the set. The number of paths from the root to TERMINAL-1 denote the number of sets stored in the ZBDD. It may be verified that the number of paths from the root to TERMINAL-1, in Figure 2.4(A) is 3, which is the cardinality of S.

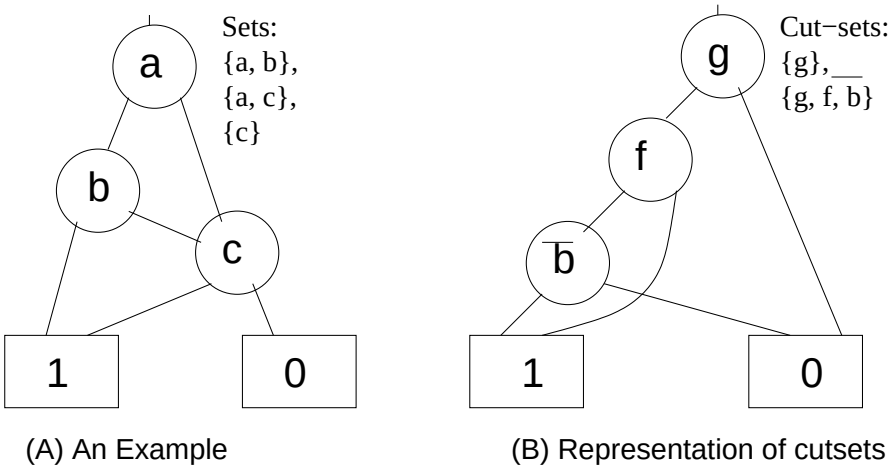


Figure 2.4: ZBDD

In our work, we use ZBDDs to store the cut-sets. Logic-0 and Logic-1 values of a gate are stored using separate variables. The cut-sets are added to the ZBDD using union operation on sets, based on the algorithm presented in [23]. The ZBDD representation of the cut-sets obtained in Figure 2.3 is shown in Figure 2.4(B). It may be noted that separate variables are used to store the positive and negative literals. For instance, $b = 0$ is represented as $\overline{b}$.

Chapter 3

Augmented Success Driven Learning

Solution subspaces heavily overlap during preimage computation. In this Chapter, we introduce the concept of *Augmented Success Driven Learning*, to *learn* from solution subspaces and prune them, based on *partially equivalent search-states*. A Mutli-Terminal ZBDD (MT-ZBDD) is used to store the solution cut-sets, obtained during ATPG. A traversal algorithm is presented to find a *superset* of the current cut-set, from the MT-ZBDD. Based on the algorithm, a path to an equivalent search-state is identified earlier in the Decision Tree. The remaining decision nodes, to reach the equivalent search-state, are identified by a simple *Upward Traversal* of the Decision Tree. These inputs are appropriately added to the Decision Tree and then the current branch is linked to the solution subspace. *Multiple-input assignment* makes use of previously encountered solution subspaces, avoids certain iterations for the decision engine of ATPG and speeds up preimage computation.

3.1 Basic Idea

Figure 3.1 shows a portion of the Decision Tree for one of the ISCAS '89 benchmark circuits obtained by a conventional PODEM algorithm [10], without *learning*. The nodes represent the decisions taken by a backtrace algorithm using Distance based Testability Measure. Let SSx, SSy, SSz represent the cut-sets at the corresponding branches in the Decision Tree, explored in the order: $SSx < SSy < SSz$.

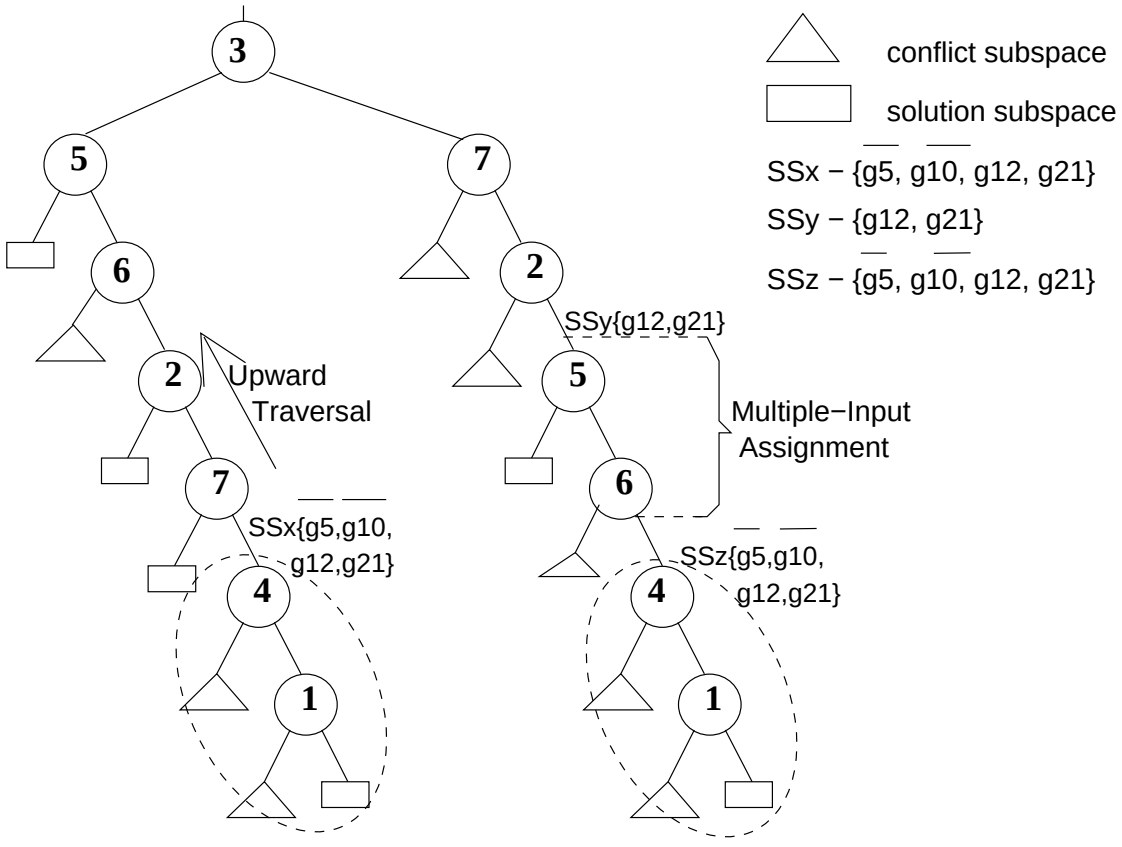


Figure 3.1: Decision Tree - Augmented Success Driven Learning

In the Figure 3.1, we see that the solution subspaces below SSx and SSz are identical. This can be *learnt* from the corresponding equivalent cut-sets. When SSz is reached, we *learn* that

SSz ≡ SSx and the current branch is simply linked to the solution subspace of SSx. This is the concept of *Success Driven Learning*, which avoids searching the same solution subspace again. It is seen that, although we are on a path to an equivalent search-state, we have to wait for an exact equivalence. However, it is possible to identify *partially equivalent cut-sets* and justify the remaining gates to obtain equivalence, based on the following observation.

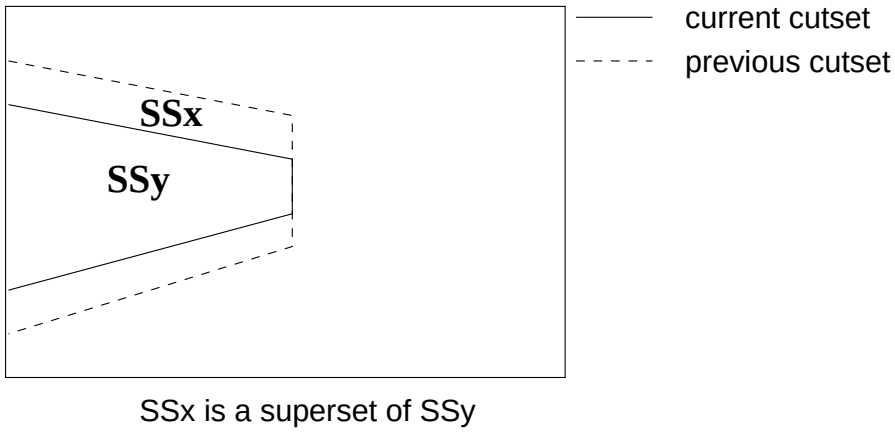


Figure 3.2: Relationship among solution cut-sets

Consider SSy as the current cut-set in the Decision Tree shown in Figure 3.1. It may be seen that SSy is a *subset* of SSx. The decisions made in between SSy and SSz justify the gates in the difference of cut-sets (SSx - SSy); i.e. $\{g5 = 0, g6 = 0\}$ justify $\{g5 = 0, g10 = 0\}$. In Figure 3.2, it is observed that the *inputs* in the difference of cones (Cone_SSx - Cone_SSy), can justify the gates in the difference of cut-sets (SSx - SSy). When SSy is reached, we *learn* that assigning $\{g5 = 0, g6 = 0\}$ leads to an equivalent cut-set, SSx. *Multiple-input assignment* is made and the current branch is linked to the solution-subspace of SSx. By forcing multiple-input assignment, the ATPG makes use of previously encountered solution subspaces. A path to equivalent solution cut-sets is identified in the Decision Tree and multiple-input assignment reduces the iterations for *backtrace()* [10] in ATPG. This effectively speeds up preimage computation.

The other branches (i.e. $g5 = 1, g6 = 1$) of the mutliple-inputs still need to be searched for potential solutions, since we do not *learn* anything about those subspaces. This maintains the *completeness* of the approach. We call this approach, *augmented success-driven learning*.

```
function preimage() {
 // step 1. Terminal Conditions
   if (objective_satisfied()) {
      update_solution_BDD();
      update_success_counter();
      return;
   }
   if (objective_violated()) return;

   // step 2. reuse knowledge learnt before
   current_cutset = find_search_state();
   result = findSupSet_sZBDD(current_cutset));
   if (result == EQUIVALENT) {
      update_solution_BDD();
      update_success_counter();
      return;
   } else if (result == SUPERSET) {
      multi_inputs();
      update_solution_BDD();
      update_success_counter();
      return;
   }
```

```
// contd...
// step 3. get new decision node
   decision_node = backtrace();
   imply();
   preimage();

// step 4. explore other branch
   flip_decision(decision_node);
   imply();
   preimage();

// step 5. store search state
   if (success_counter() > 0)
      update_sZBDD();

}
```

Figure 3.3: Algorithm - Augmented Success Driven Learning

The above idea is incorporated into PODEM algorithm as shown in Figure 3.3. It may be seen that two additional steps - Step 2 and Step 5 - are added to the algorithm. In Step 2, we search for a *superset* or an equivalent of the current cut-set, in the stored solution cut-sets. If a superset exists, we find the inputs in the difference of cut-set cones. Then the multiple-inputs are added to the Decision Tree and the current branch is linked to the corresponding solution subspace of the *superset*. If an equivalent is found, we directly link to the solution subspace and continue to search for the next solution. In Step 5, we store the solution cut-sets. A counter is defined for every node in recursion to keep track of the

number of solutions below the current node. If the counter is greater than zero, then we store the current cut-set.

3.2 Solution-ZBDD

The cut-sets in the solution branches of the Decision tree are stored in a Multi-Terminal ZBDD called the solution-ZBDD(sZBDD). The links to the solution subspaces are stored as the Terminal Nodes. The number of paths in the ZBDD corresponds to the number of solution cut-sets in the Decision Tree.

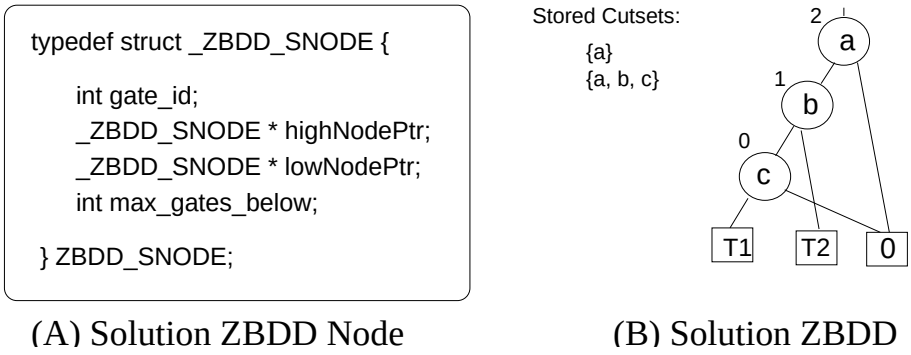


Figure 3.4: Solution-ZBDD Node

A solution-ZBDD node is defined as shown in Figure 3.4(A). The first member in the node refers to a gate in the cut-set. The second and third members are links to the children of the node. The last member identifies the 'maximum number of included nodes', below this node, of all the paths that lead to a NON-ZERO terminal. For instance, in Figure 3.4(B), max_spec_below for every node is shown. This field is updated when a new cut-set is added to the solution-ZBDD. It may be noted that the number specifies only the nodes whose 1-edge leads to a non-zero terminal.

3.2.1 Storing Solution Cutsets

A cut-set is added to the ZBDD by union operation on sets. A ZBDD is built for the cut-set to be added. An algorithm to perform union operation between two ZBDDs is presented in Figure 3.5.

```
// Add a cutset to the solution−ZBDD
function update_sZBDD(curr_cutset){

        // Step1. Convert the current cutset to ZBDD form
        curr_root = getZBDD(cutset);

        // Step 2. Add the cutset ZBDD to solutionZBDD
        return (recursive_union(curr_root, ZBDD_root));
}
```

```
// Recursive union
function recursive_union(node1, node2){

        // Step 1. Terminal conditions
        if ((node1 == node2)  || (node1 == TERMINAL_0)
            return node2;
        if (node2 == TERMINAL_0)
            return node1;

        // Step 2. Perform recursive union based on ZBDD order
        if (node1−>gate_id < node2−>gate_id){
            index = node1−>gate_id;
            highNodePtr = node1−>highNodePtr;
            lowNodePtr = recursive_union(node1−>lowNodePtr, node2);
            max_spec_below = max(node1−>max_spec_below, node2−>max_spec_below);
            return (index, highNodePtr, lowNodePtr, max_spec_below);

        else if (node1−>gate_id > node2−>gate_id){
            index = node1−>gate_id;
            highNodePtr = node2−>highNodePtr;
            lowNodePtr = recursive_union(node1, node2−>lowNodePtr);
            max_spec_below = max(node1−>max_spec_below, node2−>max_spec_below);
            return (index, highNodePtr, lowNodePtr, max_spec_below);

         } else {

            index = node1−>gate_id;
            highNodePtr = recursive_union(node1−>highNodePtr, node2−>highNodePtr);
            lowNodePtr = recursive_union(node1−>lowNodePtr, node2−>lowNodePtr);
            max_spec_below = max(node1−>max_spec_below, node2−>max_spec_below);
            return (index, highNodePtr, lowNodePtr, max_spec_below);
          }
}
```

Figure 3.5: Algorithm to add solution cut-sets

3.2.2 Search for Supersets

As explained in section 3.1, we search for a superset in the set of solution cut-sets, at every branch of the Decision Tree. This search is performed by a simple traversal of the solution-ZBDD, as shown in the algorithm in Figure 3.6.

```
// The ZBDD and the cutset are in the same order
// cs − cutset
function findSupSet_sZBDD(curr_cutset) {

    cs_ind=0; global_counter=0;
    num_csNodes=length(curr_cutset);

    // Step 1. Search the ZBDD
    link = search_sZBDD(root, cs_ind);

    // Step 2. Identify Equivalence or superset
    if(link != 0){
        if(global_counter==0)
            return EQUIVALENCE;
        else
            return SUPERSET;
    }else
        return 0;
}
```

```
function search_sZBDD(curr_node, cs_ind) {
    // Step 1. Terminal conditions
    if (cs_index>=num_csNodes && curr_node==TERMINAL_NZ)
        return link;
    if (curr_node==TERMINAL_0 || curr_node==TERMINAL_NZ)
        return 0;
    if (curr_node−>max_gates_below < num_csNodes−cs_ind)
        return 0;
    if (curr_node−>gate_id < cutset[cs_ind])
        return 0;

    // Step 2. Traversal
    if (curr_node−>gate_id == cutset[cs_ind]){
    // This node exists in the cutset
        if ((link=search_sZBDD(curr_node−>highNodePtr, ++cs_ind)>0)
            return link;
        −−cs_index;
        return 0;
    }else{
    // This node does not exist in the cutset
        if ((link=search_sZBDD(curr_node−>lowNodePtr, cs_ind)>0)
            return link;
        global_counter++;
        if ((link=search_sZBDD(curr_node−>highNodePtr, cs_ind)>0)
            return link;
        global_counter−−;
    }
}
```

Figure 3.6: Algorithm to search for a superset

Efficient heuristics are incorporated in the algorithm to find a super-set earlier, based on the following facts:

- The ZBDD is ordered and we can search for the elements in the cut-set one by one, by traversal.

- If a node exists in the current set, only the 1-branch of the node needs to be searched.

- If a node does not exist in the current set, both the branches of the node needs to be searched. The 0-branch of the node is traversed first. *This gives preference to exact matches over super-sets.*

- If all elements in the cut-set are exhausted, it means that a superset exists and we traverse to the nearest NON-ZERO TERMINAL and return the solution subspace-link.

- If 'max_spec_below' of a ZBDD node is less than the number of remaining nodes in the current set, no super-set exists below the current searching node. This step helps to avoid traversing a large number of cut-sets and hence the worst-case complexity.

An equivalent cut-set or superset is distinguished by using a *global_counter* during traversal. If we traverse the 1-branch of any ZBDD node that is not in the current cut-set, the counter is incremented as shown in Step 2 of the algorithm. The counter is decremented if both branches of that node is searched. At the end of a successful search, if the counter value is greater than zero, then it denotes superset; otherwise it denotes an equivalent set.

3.2.3 Multiple Input Assignment

When a *superset* of the current cut-set exists in the solution-ZBDD, search_sZBDD() traverses the smallest possible *superset* and returns the link to the solution subspace. The inputs required to justify the gates in the difference of cut-sets need to be identifed. They can lead to an equivalent cut-set and facilitate sharing of solution subspaces.

We know that the superset is justified by the decisions above the corresponding search-space in the Decision Tree. An *upward traversal* in the Decision Tree can identify these inputs. However, some of these inputs lie inside the cut-set cone of the current branch. These inputs are blocked by the current cut-set and may potentially justify a few gates in the current cut-set. The cut-sets at every decision branch are identified using a simple backtrace algorithm as explained in Chapter 2. Simultaneously, the unassigned inputs outside the cut-set cone are marked in the circuit. During *upward traversal*, only the marked inputs are chosen for multiple-assignment. These inputs are guaranteed to justify the gates in the difference of two cut-sets and lead to the equivalent search-state. It may be noted that the chosen inputs lie in the difference of the two cutset-cones.

3.2.4 Order of Multiple-input Assignment

For Multiple-Input Assignment in Figure 3.1 at SSy, ($x5 = 0$ and $x6 = 0$) need to be assigned. Figure 3.7 shows the Decision Tree fragments for the two possible input-orders. It is seen that the number of backtracks and number of nodes depend on the order of input assignments. This is analogous to the size of a BDD depending on the order of its variables. In most cases, multiple input-assignment in the same order as before gives good results, since it was previously determined by PODEM.

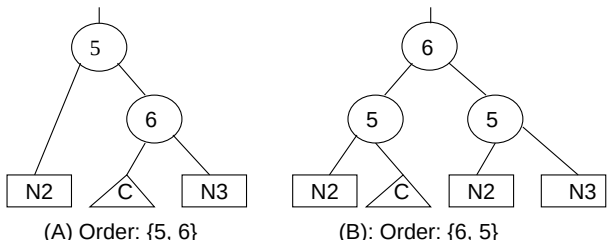


Figure 3.7: Order of Multiple Input Assignment

3.2.5 Choice of Parents

The Decision Tree becomes a Directed Acyclic Graph (DAG) due to sharing of equivalent solution subspaces. All parents of a node have a link to that node. All parents that share the same solution subspace have equivalent cut-sets. During *upward traversal* of the Decision Tree, we can traverse any one of the parents, to find multiple inputs. So, it is sufficient to store any one parent in each node. A typical Decision Tree is shown in Figure 3.8. It may be seen that cut-sets SSb and SSc share the same solution subspace.

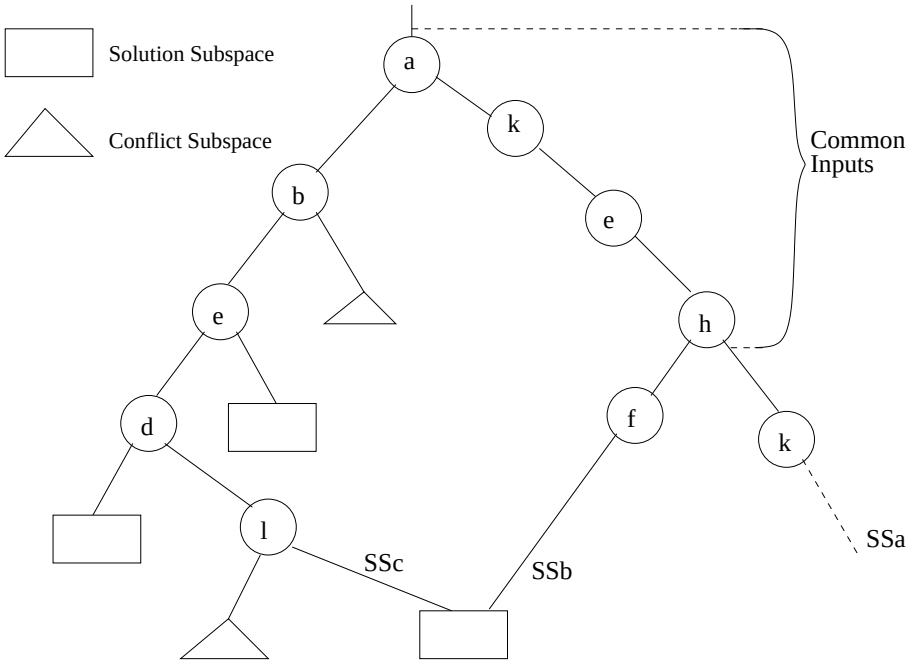


Figure 3.8: Choice of Parents

It is desirable to choose a parent that leads to lesser number of input assignments during *upward traversal*. This will help to reach the equivalent search-state faster than other parents. In the current branch, the most recent parent of the solution subspace is most likely to have maximum number of decisions common with the current search-state, due to the nature

of branching. Consequently, this will lead to a minimal number of *multiple-inputs* during upward-traversal. Based on this fact, only the most recent Parent, is stored in each of the Decision Tree nodes.

Consider the Decision Tree in Figure 3.8. Let the search-states SSa, SSb, SSc be explored in the order $SSc < SSb < SSa$ and $SSb \equiv SSc$. Our current state is at SSa and we obtained a superset match with SSb and SSc. It is seen that SSa has more Decision Tree nodes in common with SSb than SSc. If we choose SSb over SSc, then only a few inputs need to be assigned, since a major part of the Decision Tree is common for SSb and SSa. Since this situation is more likely to occur, we store only the recent parent in the Decision Tree Node for *upward traversal*.

3.2.6 Algorithm Complexity

The worst-case complexity of the proposed algorithm is $2^{n-k} + k$, where

- k is the number of elements in the current cut-set and
- n is the maximum depth of the ZBDD.

The maximum depth of the ZBDD corresponds to the longest solution cut-set obtained so far, depending on the ZBDD variable order. Experiments show that the length of cut-sets are very small compared to the number of gates in the circuit. Efficient heuristics help to avoid a large number of cut-sets and identify the superset earlier during the search.

3.3 Solution Explosion

For large designs, huge number of states are obtained that can reach the target state in one or more transitions. It becomes difficult to represent the complete preimage as an array of states, since the array may become very huge. Conventional ROBDDs potentially suffer from the Memory explosion problem. The problem of representing the set of preimage states in a compact data-structure is called *Solution Explosion* problem.

In our method, we represent the Decision Tree as a free BDD that represents the complete preimage set. Due to sharing of solution subspaces, it becomes a compact representation for the set of states. On the other hand, *Augmented Success Driven Learning* further forces the ATPG engine to make use of previously explored solution subspaces. As a result, more solution subspaces are shared that facilitate the storage of states in a more compact structure. Thus our technique inherently addresses the *Solution Explosion* problem to represent the complete preimage set. In some cases, the number of solutions found by our technique may exceed the number of solutions found by *Success driven learning* alone. However, the number of equivalent/superset solution subspaces indicate better sharing of nodes and lesser number of nodes to store the complete preimage set.

3.4 Experimental Results

The above technique is integrated into the conventional PODEM algorithm and implemented in C++. Simliarly, *Success Driven Learning* alone is implemented in PODEM algorithm. A backtrack limit of 1,000,000 is set for both the ATPG engines. Experiments are conducted on a 1.8 GHz, Pentium 4 machine with 512 MB RAM. Random conjectures of target-states are chosen for each circuit and the 1-cycle preimages for these states are found. The two techniques are compared and the results are tabulated for ISCAS '89 benchmark circuits in

Tables 3.1 and 3.2. The complete preimage set is computed for the target states.

The first column denotes the circuit. Columns 2 to 5 represent the number of solutions, number of backtracks, number of equivalent cut-sets and the time taken by *Success Driven Learning* alone. Columns 6 to 10 represent the number of solutions, number of backtracks, number of equivalent cut-sets, number of supersets and time taken for our technique.

In Table 3.1, the results for small and medium-sized circuits are tabulated. For these circuits, the time taken to compute the preimage is very less. The techniques can be compared based on the number of backtracks, number of supersets and number of equivalent cut-sets. It is seen that a considerable number of supersets occur in these circuits. There is no significant time overhead for these circuits. Due to the increased number of supersets and equivalents identified, a large number of solution subspaces are shared. This results in a compact BDD to store the preimage set. Generally, the number of backtracks decrease due to the identification of supersets. On the other hand, for circuits s526 and s953, we witness an increase in the number of backtracks. This is due to the inappropriate order of *Multi-Input Assignment*, after the identification of supersets.

The experimental results for large ISCAS '89 benchmark circuits are tabulated in Table 3.2. A large number of supersets are identified for s9234, s9234.1 and s15850. It is seen that significant savings in time is also obtained for s9342, s13207 and s15850. Similar to smaller circuits, the number of backtracks are reduced in most of the circuits. Due to the *Order of Multiple-input* assignment, it is possible that the number of backtracks increase as explained in Section 3.2.4. Supersets are identified in all the circuits, which results in more sharing of solution subspaces.

In order to verify the consistency of the approach, random target states are generated and representative results for the two techniques are tabulated in Table 3.3 for the same circuit. The experiments are conducted for the benchmark circuit - s5378. It is seen that significant

Table 3.1: Preimage Computation for ISCAS '89 circuits - I

	Success Driven Learning[17]				Augmented Success Driven Learning				
ckt	#soln	#bcktrck	#equiv	time(s)	#soln	#bcktrck	#equiv	#supersets	time(s)
s298	32	56	7	0.08	32	56	5	2	0.07
s344	22	78	10	0.07	20	71	6	1	0.04
s349	22	65	8	0.05	22	63	6	2	0.04
s382	64	53	12	0.08	64	13	13	4	0.06
s386	52	83	13	0.08	52	82	10	4	0.01
s400	40	61	13	0.06	40	46	10	1	0.07
s420.1	36	111	13	0.06	36	111	12	1	0.03
s444	20	139	10	0.07	20	140	7	3	0.06
s510	10	83	5	0.05	10	83	5	0	0.05
s526	28	101	12	0.05	28	118	9	3	0.04
s641	762	8939	188	0.24	784	9499	191	19	0.24
s713	3696	205	88	0.06	2312	185	63	17	0.02
s820	49	222	16	0.05	49	222	15	1	0.03
s832	54	225	18	0.04	54	225	15	3	0.03
s838.1	3320	239	32	0.06	1720	246	37	6	0.03
s953	5852	4030	1556	0.24	6032	4104	1432	238	0.28
s1196	35	158	22	0.04	35	151	20	2	0.01
s1238	118	275	88	0.05	118	275	88	0	0.04

Table 3.2: Preimage Computation for ISCAS '89 circuits - II

	Success Driven Learning[17]				Augmented Success Driven Learning				
ckt	#soln	#bcktrck	#equiv	time(s)	#soln	#bcktrck	#equiv	#supersets	time(s)
s1423	10.3M	121298	1634	4.92	8.4M	120318	1548	53	5.86
s1488	43	204	10	0.08	44	204	11	1	0.02
s1494	57	116	14	0.07	57	115	13	1	0.04
s5378	1.9M	303378	291	17.41	1.8M	303313	246	15	13.09
s9234	1.6G	272229	38219	73.71	2.9M	173289	32446	8863	44.82
s9234	1.2G	143460	13478	20.01	238M	132637	10469	5023	20.19
s9234.1	135M	48680	7388	7.12	2.2G	48179	6698	1213	6.34
s9234.1	2.7G	10127	1411	1.62	1.1M	10951	1745	448	2.58
s13207	355M	738867	2056	106.43	3.6G	742361	3096	356	76.97
s13207.1	1.8M	427680	1028	58.44	1.8M	428261	1296	323	42.52
s15850	1.3M	57029	26115	17.46	3.8M	62953	25945	4215	16.25
s15850.1	45M	31134	4014	7.72	50M	31883	4029	22	6.61
s35932	1.3G	26986	100	6.29	1.3G	27002	96	4	3.49
s38417	1.9G	1769	435	1.17	1.3G	3650	415	139	1.49

improvement in time is obtained for the first three target-states. The results are in line with previous observations and supersets are obtained for most of the target states.

Table 3.3: Preimage Computation for 10 targets of s5378

	Success Driven Learning [17]				Augmented Success Driven Learning				
S.No	#soln	#bcktrck	#equiv	time(s)	#soln	#bcktrck	#equiv	#supersets	time(s)
1.	1.9M	303378	291	17.41	1.8M	303313	246	15	13.09
2.	34K	153183	211	8.75	43K	153515	221	16	6.19
3.	1.3G	15079	1243	1.48	1.3G	15077	1240	2	1.16
4.	9.9G	13468	7746	1.29	8.7G	17623	10287	1002	1.54
5.	62M	4899	2817	0.55	62M	4602	2318	202	0.43
6.	856M	603	375	0.15	755M	610	302	25	0.15
7.	58K	26981	295	1.69	58K	26211	209	38	1.40
8.	65M	4899	2817	0.55	65M	4602	2318	202	0.43
9.	1.6G	655	287	0.14	1.6G	562	245	6	0.13
10.	4.05G	2917	1239	0.39	1.9G	4652	1650	394	0.37

Chapter 4

Conflict Driven Learning

In this chapter, we explain the concept of *Search-State based Conflict Driven Learning*. Unlike conventional conflict-driven techniques, we learn from the search-states of the circuit. As a dual of pruning solution subspaces, the conflict subspaces in the Decision Tree are pruned, by learning from the conflict cut-sets. Cut-sets in the conflict branches of the Decision Tree are stored in a ZBDD. Unlike the *search problem* in solution-ZBDD, we encounter a *decision problem* in case of a conflict-ZBDD. A similar traversal algorithm is proposed for efficient set manipulation. Finally, experimental results are presented that demonstrate many conflict subspaces occur during preimage computation and the proposed technique prunes significant number of conflict subspaces.

4.1 Basic Idea

Consider a portion of the Decision Tree shown in Figure 4.1. It represents preimage computation for one of the ISCAS '89 benchmark circuits, obtained by a conventional PODEM algorithm [10], without learning. In the Figure 4.1, let SSa and SSb represent the cut-sets

at the corresponding branches in the Decision Tree, explored in that order.

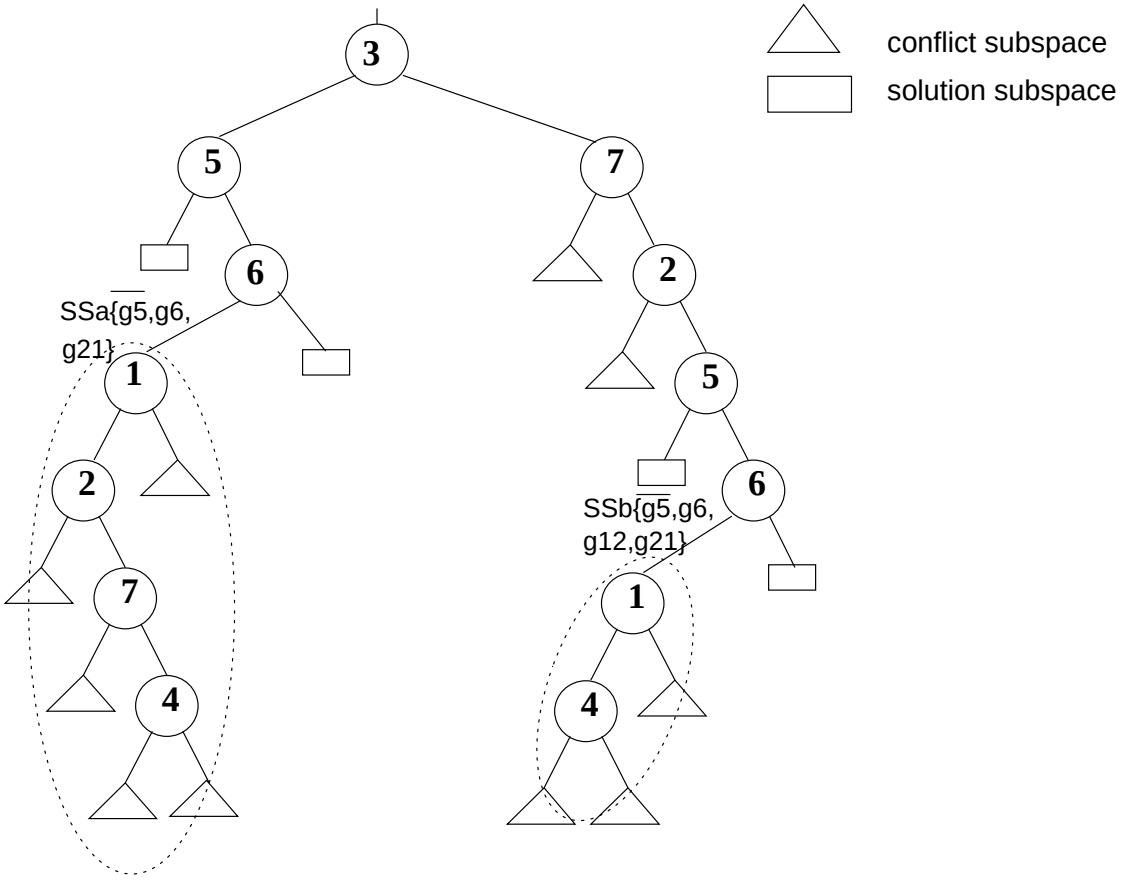


Figure 4.1: Decision Tree

Figure 4.1 shows that the subspace below SSa is a conflict subspace. In the corresponding circuit in Figure 4.2, it can be argued that all possible input assignments, after cut-set SSa is reached, will lead to a conflict. Any assignment to the gates outside the cone of SSa will still lead to a conflict. When an equivalent or a superset of SSa is encountered, we can backtrack, since the search-subspace below the current branch is guaranteed to be a conflict subspace. For instance, it can be verified in the Decision Tree that SSb is a *superset* of SSa and leads to a conflict subspace. Considering SSb as the current cut-set, we *learn* that a *subset* of

the current cut-set previously lead to a conflict. We can immediately backtrack and avoid searching a conflict subspace. We call this *search-state based conflict-driven learning*.

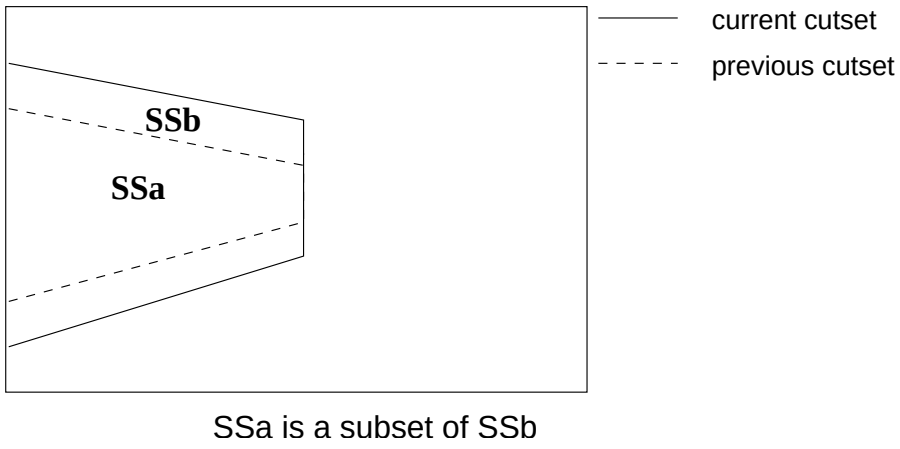


Figure 4.2: Relationship among conflict cut-sets

The idea is incorporated into a recursive PODEM algorithm as shown in Figure 4.3. At every branch of the Decision Tree, we search the stored conflict-cutsets for *subsets* or equivalents. If one exists, we *learn* that the current branch will lead to a conflict subspace and immediately backtrack to prune the conflict region in the search-space. All the cut-sets in the conflict branches of the Decision Tree are stored in a conflict-ZBDD as shown in Step 5 of the algorithm. Before exploring a search-space, we search the stored conflict cut-sets for *subset or equivalent* in Step 2 of the algorithm. Unlike *Augmented Success Driven Learning*, we are not concerned with the link to the Decision Tree. This is a *decision problem* and we can backtrack when we find a previous conflict subset.

In previous conflict driven techniques, the *cause* for a conflict is analyzed. Based on the *cause*, extra information is added to the ATPG engine to avoid reaching the same conflict again. In these techniques, there is an extra overhead to analyze the conflicts and store the conflict-information. On the other hand, we use the cut-set as a representation for a conflict subspace. This helps to prune many conflicts and hence is more powerful than conventional

```
function preimage() {
 // step 1. Terminal Conditions
    if (objective_satisfied()) {
       update_solution_BDD();
       update_success_counter();
       return;
       }
   if (objective_violated()) return;
 // step 2. reuse knowledge learnt before
    current_cutset = find_search_state();
    if (subSetExists_in_cZBDD(current_cutset))
       return;
 // step 3. get new decision node
    decision_node = backtrace();
    imply();
    preimage();
 // step 4. explore other branch
    flip_decision(decision_node);
    imply();
    preimage();
 // step 5. store search state
    if (success_counter() == 0)
         update_cZBDD();
}
```

Figure 4.3: Algorithm - Conflict Driven Learning

conflict driven techniques. Note that this *learning* is different from conventional conflict analysis, since we learn from the search-states during search.

4.2 Conflict-ZBDD

Cut-sets in the conflict branches of the Decision Tree are stored in a conflict-ZBDD (cZBDD). The ZBDD nodes represent the cut-set elements and a path to TERMINAL-1 denotes a conflict cut-set in the Decision Tree. The number of paths in the ZBDD represents the

number of conflict cut-sets encountered during ATPG.

A conflict-ZBDD node is defined as shown in Figure 4.4. The first three members are similar to that of solution-ZBDD. The last member identifies the 'minimum number of included nodes' below the node, of all the paths that lead to TERMINAL-1 in the ZBDD. Similar to solution-ZBDD, this field is updated when a new cut-set is added to the conflict-ZBDD.

```
typedef struct _ZBDD_CNODE {
    int gate_id;
    _ZBDD_CNODE * highNodePtr;
    _ZBDD_CNODE * lowNodePtr;
    int min_gates_below;
} ZBDD_CNODE;
```

Figure 4.4: Conflict-ZBDD Node

4.2.1 Algorithm

Each cut-set in the conflict branch of the decision tree is added to the ZBDD by a simple union operation [23]. The algorithm to store a cut-set in the conflict-ZBDD is same as that of a solution-ZBDD, as shown in Section 3.2.1. The algorithms proposed in [23] were not suitable to find the existence of a subset. This is a decision problem and the algorithms in [23] aim at computing sets. We propose a simple traversal algorithm as shown in Figure 4.5 to determine if a *subset* exists or not. The members of each node and the current cut-set facilitate efficient traversal of the ZBDD.

The algorithm is an exact dual to that of solution-ZBDD and hence the following heuristics

```
// The ZBDD and the cutset are in the same order
// cs − cutset
function subSetExists_in_cZBDD(curr_cs) {
  cs_ind=0; num_csNodes=length(curr_cs);

   // step 1. Discard unnecessary gates
   while (curr_cs[++cs_ind] < root−>gate_id);

   // step 2. Search the ZBDD
   return (search_cZBDD(root, cs_ind));
}
```

```
function search_cZBDD(curr_node, cs_ind) {
    // step 1. Terminal conditions
      if (curr_node == TERMINAL_1)      return true;
      if (curr_node == TERMINAL_0)      return false;
      if (curr_node−>min_gates_below > num_csNodes−cs_ind)
        return false;
    // step 2. Traversal for subset
      if (curr_node−>gate_id == curr_cs[cs_ind]){
    // This node exists in the cutset
        temp_ind=cs_ind; temp_root=curr_node−>highNodePtr;
        while (curr_cs[++cs_ind] < temp_root−>gate_id);
        if (search_cZBDD(temp_root, cs_ind))
            return true;
        cs_ind=temp_ind; temp_root=curr_node−>lowNodePtr;
        while (curr_cs[++cs_ind] < temp_root−>gate_id);
        if (search_cZBDD(temp_root, cs_ind))
            return true;
        cs_ind = temp_ind;
      }else  // This node is absent in the cutset
        return (search_cZBDD(curr_node−>lowNodePtr, cs_ind));
 }
```

Figure 4.5: Algorithm to find a subset

are incorporated:

- The ZBDD is ordered and we can search for the elements in the cut-set one by one, by traversal.

- If a node does not exist in the current set, only the 0-branch of the node needs to be searched.

- If a node exists in the current set, both branches of the node needs to be traversed. The 1-branch of the node is traversed first. *This gives preference to exact matches over sub-sets.*

- If we reach the 1-TERMINAL during traversal, it means that a subset exists and we return SUCCESS.

- If the 'minimum number of specified nodes' below the current ZBDD node is greater than the remaining elements in the current set, no sub-set exists below the current-node. This step helps to avoid traversing a large number of cut-sets and to avoid the worst-case complexity.

4.2.2 Algorithm Complexity

The worst-case complexity of the proposed algorithm is $2^k + n - k$, where

- k is the number of elements in the current cut-set and
- n is the maximum depth of the ZBDD

It may be noted that k is usually very small, when compared to the number of gates in the circuit. Hence the subsets can be identified very early. Efficient heuristics facilitate faster traversal of the ZBDD and early identification of the *subset*, if one exists. The members of the ZBDD node and the ordered elements of the cutset facilitate faster traversal and identification of subsets.

4.3 Experimental Results

The proposed approach helps search-space pruning, only if large number of conflict *subsets* occur during preimage computation. In order to assess the occurence of conflict subspaces, it is necessary to conduct experiments and verify the requirement of the approach. *Search state based conflict driven learning* alone is integrated with PODEM as shown in Algorithm 4.3. The algorithm is implemented in C++ and experiments are conducted on a 1.8 GHz, P4 machine with 512 MB RAM operating on Linux. A backtrack limit of 1,000,000 is set for the ATPG engine. Similar to the previous experiments, random conjectures of target-states are chosen for ISCAS '89 benchmark circuits and the results are compared with *Success Driven Learning* alone.

Table 4.1 lists the results for small and medium-sized circuits. The first column lists the name of circuits. Columns 2-5 report the number of solutions, backtracks, conflicts, equivalent cut-sets obtained by the *Success Driven Learning* alone. Column 6 shows the time taken for the ATPG engine to compute the preimage. Columns 6-9 report the number of solutions, backtracks, conflicts and subsets obtained by *conflict driven learning* technique. The last column reports the time taken by our technique to compute the preimage for the circuit.

It is seen that conflict subspaces occur in most of the circuits. Significant number of conflict *subsets* are identified in circuits - *s444* and *s953*. Similarly equivalent solution subspaces are present in these circuits. The solution subspaces also contain some conflict branches. As a result, equivalent solution subspaces inherently prune some of the conflict subspaces. This is reflected in the number of backtracks for the circuits. It may be seen that the number of backtracks, for circuits with more equivalent solution subspaces, is lesser in the previous technique than our technique.

The results for large circuits is tabulated in Table 4.2. Conflict subspaces overlap heavily for

unjustifiable target states. For these targets, *success driven learning* aborts and significant speed up is obtained by adding *conflict driven learning*. On the other hand, for targets, with huge number of solutions, *success driven learning* obviously outperforms *conflict-driven learning*. It is shown that this phenomenon is common to most of the ciruits. Results for large circuits, demonstrate the necessity to integrate both success-driven and conflict-driven techniques in the ATPG engine. A symmetric treatment of both success and conflict-driven techniques is required to *learn* from the solution and conflict subspaces in the Search-Space.

Experimental results for 10 random targets of s5378 are tabulated in Table 4.3. It may be generally seen that the number of conflicts are more in some of the circuits as compared to the previous technique. In *success driven learning* solution subspaces are pruned, which may include conflicts. As a result, inherently some of the conflicts are also pruned. In *conflict-driven learning* alone, we depend only on conflict subspaces that do not contain solution subspaces. As a result, there is a necessity to integrate both these approaches for a symmetric treatment of solution and conflict subspaces.

Table 4.1: Preimage Computation for ISCAS '89 circuits - I

	Success Driven Learning [17]					Conflict Driven Learning				
ckt	#soln	#bcktrck	#conf	#equiv	time(s)	#soln	#bcktrck	#conf	#subsets	time(s)
s298	28	154	133	17	0.06	28	193	160	6	0.02
s344	9	49	41	6	0.05	9	63	55	0	0.06
s349	14	155	149	5	0.05	14	110	87	10	0.02
s382	21	53	46	5	0.04	21	79	55	4	0.04
s386	36	62	30	7	0.04	36	71	34	2	0.03
s400	81	73	55	15	0.05	81	383	283	20	0.04
s420.1	24	86	77	9	0.06	24	227	182	22	0.03
s444	41	179	158	20	0.07	41	277	176	61	0.04
s510	10	83	74	5	0.09	10	88	76	3	0.07
s526	45	141	126	11	0.05	45	269	211	14	0.08
s641	None	557	558	0	0.07	None	192	162	31	0.04
s713	200	112	78	34	0.06	200	808	570	39	0.02
s820	50	223	179	16	0.10	50	192	127	16	0.04
s832	8	167	161	2	0.06	8	127	103	17	0.05
s838.1	54	117	103	14	0.06	54	496	419	24	0.05
s953	52	1154	1119	43	0.15	52	887	700	136	0.11
s1196	75	412	353	39	0.07	75	490	400	16	0.04
s1238	101	423	360	59	0.05	101	742	612	30	0.04

Table 4.2: Preimage Computation for ISCAS '89 circuits - II

	Success Driven Learning [17]					Conflict Driven Learning				
ckt	#soln	#bcktrck	#conf	#equiv	time(s)	#soln	#bcktrck	#conf	#subsets	time(s)
s1423	None	2424	2425	0	0.32	None	2012	1879	134	0.25
s1488	43	204	175	10	0.08	43	194	129	23	0.03
s1494	5	110	106	4	0.07	5	87	73	10	0.03
s5378	Abt	1M	1M	0	56.90	None	1K	698	673	0.13
s5378	40K	167	101	66	0.08	40K	194K	145K	8K	5.45
s9234	None	124K	124K	0	27.03	None	509	191	319	0.19
s9234	236K	1.4K	992	399	0.32	236K	806K	561K	8K	38.62
s9234.1	Abt	1M	1M	0	290.96	None	13K	9K	4770	5.09
s9234.1	469M	2K	988	1222	0.43	287K	1M	549K	162K	55.06
s13207	None	40981	40K	0	11.13	None	730	448	283	0.33
s13207	25K	72	35	37	0.16	25K	109K	84K	0	7.51
s13207.1	None	1.3K	136K	0	2.92	None	126	73	54	0.12
s13207.1	133K	146	73	71	0.18	133K	258K	125K	0	16.21
s15850	29K	340	260	80	0.31	29K	111K	81K	0	10.11
s15850.1	None	26K	267K	0	13.66	None	11K	11K	289	6.17
s15850.1	1.6M	4K	4K	754	1.32	160K	1M	426	0	139.36
s35932	65K	112	59	53	0.26	65K	345K	279K	0	38.68
s38417	Abt	1M	1M	0	2153.13	None	25K	21K	4K	60.99
s38417	3G	782	115	666	0.79	315K	1M	683K	1K	185.56

Table 4.3: Preimage Computation for 10 targets of s5378

	Success Driven Learning [17]					Conflict Driven Learning				
S.No	#soln	#bcktrck	#conf	#equiv	time(s)	#soln	#bcktrck	#conf	#subsets	time(s)
1.	Abt	1M	1M	0	87.56	None	11K	5.8K	5.3K	1.30
2.	None	10K	10K	0	1.62	None	757	652	106	0.14
3.	None	11K	11K	0	1.30	None	1.2K	1.1K	93	0.16
4.	Abt	1M	1M	0	95.40	None	529	273	257	0.09
5.	None	354K	354K	0	25.95	None	791	434	358	0.11
6.	6.9K	1087	925	162	0.20	6.9K	450K	377K	3.5K	13.01
7.	3G	4.3K	2.9K	1480	0.54	142K	1M	857K	0	27.26
8.	Abt	1M	1M	0	54.71	None	3.3K	1.6K	1.7K	0.40
9.	10M	743	648	95	0.12	199K	1M	800K	97	27.17
10.	Abt	1M	1M	0	56.90	None	1.3K	698	673	0.13

Chapter 5

Symmetric Learning

We saw that solution and conflict subspaces heavily overlap during preimage computation. It is necessary to *learn* from both subspaces for faster preimage computation. In this chapter, we integrate *Augmented Success Driven Learning* and *Search-State based Conflict Driven Learning* to provide a *Symmetric treatment* for both subspaces during the search.

5.1 Algorithm

Both the *learning* techniques are incorporated into a PODEM algorithm [10] as shown in Figure 5.1. Cutsets in the solution branches are stored in a solution-ZBDD (sZBDD) and those in the conflict branches are stored in a conflict-ZBDD (cZBDD). Two distinct features of our contribution are in Step 2. In Step 2a, we search the conflict-ZBDD database for *equivalent or subset* of current search-state. Likewise, Step 2b searches the solution-ZBDD database for *equivalent or superset* of current search-state. In Step 5, we store the cut-set in the solution-ZBDD, if the subspace below has a solution. Otherwise the cut-set is stored in the conflict-ZBDD.

```
function preimage() {
 // step 1. Terminal Conditions
   if (objective_satisfied()) {
     update_solution_BDD();
     update_success_counter();
     return;
   }
  if (objective_violated()) return;
 // step 2. reuse knowledge learnt before
 // step 2a. search stored conflict cutsets
   current_cutset = find_search_state();
   if (subSetExists_in_cZBDD(current_cutset))
     return;
 // step 2b. search stored solution cutsets
   result = findSupSet_sZBDD(current_cutset));
   if (result == EQUIVALENT) {
     update_solution_BDD();
     update_success_counter();
     return;
   }
// Contd...
```

```
 else if (result == SUPERSET) {
   multi_inputs();
   update_solution_BDD();
   update_success_counter();
   return;
 }

 // step 3. get new decision node
   decision_node = backtrace();
   imply();
   preimage();

 // step 4. explore other branch
   flip_decision(decision_node);
   imply();
   preimage();

 // step 5. store search state
   if (success_counter() > 0)
     update_sZBDD();
   else
     update_cZBDD();
 }
```

Figure 5.1: Algorithm - Symmetric Learning

5.2 Experimental Results

The Experimental Results for ISCAS '89 benchmark circuits are tabulated in Tables 5.1, 5.2 and 5.3. The experiments are conducted in the same environment as before. The first column denotes the target state, for the particular circuit. The total number of solutions, backtracks, conflicts, equivalents obtained for *success-driven* learning are reported in columns 2-5. Column 6 shows the time-taken by the ATPG engine for previous technique. Columns 7-12 report the number of solutions, backtracks, conflicts, equivalents, supersets, subsets and time taken by our technique. We can observe that supersets and subsets are obtained for most of the circuits, which demonstrates the necessity to learn from them.

The number of backtracks are generally reduced for most of the target states in smaller circuits. For larger circuits, a significant improvement in time is obtained for s5378, s9234 and s15850.1. The reduction in number of backtracks show that more than 90% of the search space is pruned for these circuits. The presence of many *equivalents* and *supersets*, for these circuits is interpreted as the sharing of many nodes in the preimage set. As a result, the preimage can be represented in a more compact manner.

Experimental results for 10 random targets of s5378 are reported in Table 5.3. The backtracks are reduced significantly for most of the target states, eg: 4, 9, 10, 11. Significant savings in time is obtained for targets - 9, 10, 11. For target state 6, we witness an increase in the number of backtracks for our techniques. This is due to the *order of multiple-input assignment* discussed in Section 3.2.4. However, the overhead in using the proposed techniques is generally low as seen in most of the target states.

Table 5.1: Preimage Computation for ISCAS '89 circuits - I

	Success-Driven [17]					Symmetric Learning						
ckt	#sln	#btrck	#conf	#eqv	t(s)	#sln	#btrck	#conf	#eqv	#sps	#sbs	t(s)
s298	28	154	133	17	0.08	28	128	99	19	1	5	0.01
s344	5	51	47	3	0.08	5	43	36	3	0	3	0.04
s349	22	157	147	4	0.07	20	74	56	2	1	11	0.06
s382	30	50	39	10	0.05	30	40	31	7	1	0	0.03
s386	29	35	13	5	0.05	29	35	13	5	0	0	0.06
s400	81	73	55	15	0.05	81	69	47	15	0	4	0.02
s420.1	20	93	81	11	0.07	17	97	79	6	6	5	0.02
s444	9	87	81	4	0.07	5	76	69	2	2	3	0.01
s510	20	116	106	7	0.06	20	106	90	6	2	5	0.02
s526	45	141	126	11	0.03	45	113	85	10	1	13	0.01
s641	98	198	164	34	0.05	92	186	154	27	3	2	0.03
s713	64	854	835	19	0.07	64	303	234	19	0	50	0.02
s820	50	223	179	16	0.06	49	161	106	15	1	12	0.05
s832	8	167	161	2	0.06	8	117	95	2	0	16	0.03
s838.1	54	117	103	14	0.08	27	164	138	8	12	6	0.01
s953	52	1154	1110	43	0.11	54	725	558	43	1	122	0.09
s1196	75	412	353	39	0.05	74	386	314	37	1	14	0.04
s1238	101	423	360	59	0.06	101	376	297	57	2	16	0.02

Table 5.2: Preimage Computation for ISCAS '89 circuits - II

	Success-Driven [17]					Symmetric Learning						
ckt	#sln	#btrck	#conf	#eqv	t(s)	#sln	#btrck	#conf	#eqv	#sps	#sbs	t(s)
s1423	513M	37K	34K	3.1K	1.90	494M	7.7K	3.6K	3.3K	83	663	0.42
s1488	43	204	175	10	0.05	44	154	106	11	1	18	0.02
s1494	5	110	106	4	0.08	5	80	67	4	0	9	0.04
s5378	96K	38K	38K	209	2.70	77K	1.2K	756	213	9	265	0.13
s9234	Abt	1M	1M	0	94.68	None	580	171	0	0	410	0.16
s9234.1	3.7G	62K	15K	47K	16.36	3.8G	57K	11K	37K	9K	617	12.33
s13207	47K	2K	18K	190	0.51	48K	1K	789	183	41	25	0.26
s13207.1	3120	78	37	41	0.18	3312	74	25	39	8	2	0.09
s15850	3.5G	2.7K	911	1807	1.18	3.3G	2.9K	984	1.9K	67	6	0.99
s15850.1	3.9G	71.5K	70K	1494	12.03	686M	2.5K	461	1.7K	148	96	0.59
s35932	83	75	55	20	0.25	83	75	55	18	2	0	0.16
s38417	None	26K	26K	0	48.54	None	141	76	0	0	66	0.56

Table 5.3: Preimage Computation for 10 targets of s5378

	Success-Driven [17]					Symmetric Learning						
ckt	#sln	#btrck	#conf	#eqv	t(s)	#sln	#btrck	#conf	#eqv	#sps	#sbs	t(s)
1.	153M	2.9K	2029	950	0.36	153M	2K	1047	850	31	87	0.18
2.	972	45	25	20	0.08	972	45	25	20	0	0	0.09
3.	10M	231K	230K	180	15.17	10M	523	184	182	2	155	0.09
4.	995K	1238	1043	195	0.18	995K	375	146	195	4	30	0.07
5.	163K	636	202	433	0.12	203K	1.1K	264	744	74	0	0.15
6.	61M	332	161	171	0.13	61M	287	107	171	0	9	0.06
7.	59M	730	523	207	0.15	67M	455	231	199	8	17	0.08
8.	4.7M	998K	997K	281	50.37	4.7M	1.3K	868	261	20	181	0.11
9.	447M	551K	548K	3.5K	31.58	432M	6.9K	1.7K	3.6K	265	1237	0.58
10.	None	121K	121K	0	10.98	None	867	265	0	0	603	0.12

Chapter 6

Conclusion

6.1 Conclusion

We presented a framework for *ATPG based preimage computation*. Conventional ROBDD based methods were sensitive to the memory limitation problem and cannot be used for larger circuits. It was shown that ATPG based methods overcome the memory limitation problem and results for large circuits were presented. Efficient learning techniques were presented to overcome the inherent time limitation of ATPG.

We identified that both solution and conflict subspaces overlap during *ATPG based preimage computation*. Efficient search-state based techniques were presented to *learn* from the solution and conflict subspaces during preimage computation. We learn from *solution supersets* to make use of previously identified solutions. *Multiple input assignment* is forced to reach an *equivalent search-state* and a previously explored solution subspace is pruned. We presented a technique to identify the *multiple-inputs* necessary to reach an equivalent search-state by *Upward Traversal* of the Decision Tree. We showed that the order of multiple-input assignment affects the performance of ATPG. We use the order previously generated by PODEM

to obtain better results. On the other hand, we learn from *conflict subsets* to make use of previously encountered conflicts. Early backtracking helps to avoid large conflict subspaces during the search. Integrating both *success driven* and *conflict driven* learning showed that our techniques prune upto 90% of the search space and computes preimage for states where a state-of-art learning technique failed.

The search states were stored in ZBDDs and new algorithms were presented to identify *supersets/subsets* in solution/conflict-ZBDD. Efficient heuristics were integrated into the algorithm for early identification of subsets/supersets. The ZBDD-based techniques were integrated into a conventional ATPG engine to generate all the solutions. The set of solutions form the complete preimage for the target state at the end. Experimental results showed that only a small overhead is incurred in implementing the proposed techniques.

6.2 Future Work

The cut-sets in all branches of the Decision Tree are stored in ZBDDs. However, all the cut-sets are not useful to the ATPG engine. It is necessary to identify cut-sets that are likely to be useful in the future and store them in the knowledge-base. This will lead to a more compact ZBDD and also reduce the time taken to search for *subsets* or *supersets*.

Preimage computation is a basic step in Model Checking and Equivalence Checking applications. In our method, the preimage set may be represented as a Reduced Boolean Circuit instead of a free BDD. It can act as a monitor circuit for the previous time-frame of the Iterative Logic Array (ILA) model of the sequential circuit. In this way, the proposed method can be extended to multiple timeframes with very little overhead.

In SAT, the array representation of clauses consumes huge memory for large CNFs. In [25, 24], Aloul et al. and Cathalic et al. studied the suitability of ZBDDs to represent

CNFs and proposed elegant algorithms for Boolean Constraint Propagation (BCP). Our techniques can be extended to ZBDD-based SAT by representing search-states as ZBDD nodes. Since ZBDDs perform set-operations efficiently, *equivalents, subsets and supersets* can be easily identified. It may be noted that no extra overhead is required in identifying the search-states and storing them.

Considerable work has been done in the field of identifying suitable Testability Measures for an ATPG engine. Many heuristics were presented to identify a solution earlier during ATPG. However, all these heuristics aim at identifying a single solution faster. In preimage computation, we aim at finding all the solutions for the target state. Conventional Testability Measures are local to each solution. New Testability Measures are required to compute the complete set of solutions faster. No work has been done so far in this direction.

During preimage computation, *Equivalent Search States* will definitely lead to identical solution subspaces. However, the vice-versa is not true. It is seen that identical solution subspaces exist for search-states that are not equivalent. Since these subspaces are not pruned by *success-driven* learning, it is necessary to identify these search-states and further speed up preimage computation.

None of the ideas discussed above have been explored yet, and hold good potential for future research.

Bibliography

[1] S. Huang and K. T. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publishers, 1998.

[2] J. M. Burch, E. M. Clarke, and D. E. Long, "Representing Circuits More Efficiently in Symbolic Model Checking," in *Proceedings of Design Automation Conference*, pp. 403–407, 1991.

[3] N. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli, "Partitioned ROBDDs: A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions," in *Proceedings of International Conference on Computer Aided Design*, pp. 547–554, 1996.

[4] P. A. Abdullah, P. Bjesse, and N. Een, "Symbolic Reachability Analysis based on SAT solvers," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Vol. 1785 of Lecture Notes in Computer Science*, pp. 124–138, 2000.

[5] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta, "Combining Decision Diagrams and SAT procedures for efficient Symbolic Model Checking," in *Proceedings of International Conference on Computer Aided Verification, Vol. 1855 of Lecture Notes in Computer Science*, pp. 124–138, 2000.

[6] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and A. Gupta, "SAT based Image Computation with Application in Reachability Analysis," in *Proceedings of Formal Methods in Computer Aided Design*, 2000.

[7] K. L. McMillan, "Applying SAT methods in unbounded Symbolic Model Checking," in *Proceedings of International Conference on Computer Aided Verification, Vol. 2404 of Lecture Notes in Computer Science*, pp. 250–264, 2002.

[8] A. Gupta, A. Gupta, Z. Yang, and P. Ashar, "Dynamic Detection and Removal of Inactive Clauses in SAT with Application in Image Computation," in *Proceedings of Design Automation Conference*, pp. 536–541, 2001.

[9] D. G. Saab, J. A. Abraham, and V. M. Vedula, "Formal Verification Using Bounded Model Checking: SAT Versus Sequential ATPG Engines," in *Proceedings of VLSI Design*, pp. 243–248, 2003.

[10] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. IEEE Press, 1990.

[11] M. H. Schulz, E. Trischler, and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation," in *IEEE Transactions on Computer Aided Design, Vol.7, No.1*, pp. 126–137, 1988.

[12] J. P. Marques-Silva and K. A. Sakallah, "Dynamic Search-Space Pruning Techniques in Path Sensitization," in *Proceedings of Design Automation Conference*, pp. 705–711, 1994.

[13] J. P. Marques-Silva and K. A. Sakallah, "A Search Algorithm for Propositional Satisfiability," in *IEEE Transactions on Computers, Vol. 48, No. 5*, pp. 506–521, 1999.

[14] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability," in *Proceedings of International Conference on Computer Aided Design*, pp. 279–285, 2001.

[15] C. Wang, S. M. Reddy, I. Pomeranz, L. Xiang, and J. Rajski, "Conflict driven techniques for improving deterministic test pattern generation," in *Proceedings of International Conference on Computer Aided Design*, pp. 87–93, 2002.

[16] M. K. Iyer, G. Parthasarathy, L. C. Wang, and K. T. Cheng, "On the development of ATPG based Satisfiability Checker," in *Proceedings of Microprocessor Test and Verification Workshop, IEEE*, 2002.

[17] S. Sheng and M. S. Hsiao, "Efficient PreImage Computation Using a Novel Success-Driven ATPG," in *Proceedings of Design Automation and Test Conference in Europe*, pp. 822–827, 2003.

[18] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD package," in *Proceedings of Design Automation Conference*, pp. 40–45, 1990.

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

[20] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT procedures instead of BDDs," in *Proceedings of Design Automation Conference*, pp. 317–320, 1999.

[21] H. J. Kang and I. C. Park, "SAT-based Unbounded Model Checking," in *Proceedings of Design Automation Conference*, 2003.

[22] I. Hamzaoglu and J. H. Patel, "New Techniques for Deterministic Test Pattern Generation," in *Proceedings of VLSI Test Symposium*, pp. 446–452, 1998.

[23] S. Minato, "Zero-suppressed BDDs and their applications," in *Proceedings of Software Tools for Technology Transfer*, pp. 156–170, 2001.

[24] F. A. Aloul, M. N. Mneimneh, and K. A. Sakallah, "Search-based SAT using zero-suppressed BDDs," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, p. 1082, 2002.

[25] P. Cathalic and L. Simon, "Multi-resolution on compressed sets of clauses," in *Proceedings of International Conference on Tools with Artificial Intelligence*, pp. 2–10, 2000.

[26] S. Padmanabhan, M. K. Michael, and S. Traqoudas, "Exact Path Delay fault coverage with fundamental ZBDD operations," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, Issue 3*, pp. 305–316, 2003.

[27] J. Giraldi and M. L. Bushnell, "EST: The New Frontier in Automatic Test Pattern Generation," in *Proceedings of Design Automation Conference*, pp. 667–672, 1990.

[28] T. Fujino and H. Fujiwara, "An Efficient Test Generation Algorithm Based on Search State Dominance," in *Proceedings of Fault Tolerant Computing*, pp. 33–40, 1992.

Vita

Kameshwar Chandrasekar was born in Chennai, a metropolitan city in India. He did his early education in Chennai. He joined Government College of Technology, Coimbatore to obtain his technical education in Electronics and Instrumentation Engineering. He obtained his Bachelor of Engineering (BE) degree in 2000. He then worked as a Software Engineer at iNautix Technologies, India for a year. He joined Virginia Tech in Fall 2001 for his Masters in Electrical Engineering. He joined Dr. Hsiao and his research group in January 2002 and since then been involved in research related to ATPG and Model Checking. His technical interests include VLSI Testing and Formal Verification Methods.