Abstraction Guided Semi-formal Verification

by

Ankur Nimish Parikh

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical and Computer Engineering

Michael S. Hsiao

Sandeep Shukla

Patrick Schaumont

June 15, 2007

Blacksburg, Virginia

Keywords: formal, verification, simulation, abstraction, preimage

Abstraction Guided Semi-formal Verification

Ankur Nimish Parikh

ABSTRACT

Abstraction-guided simulation is a promising semi-formal framework for design validation in which an abstract model of the design is used to guide a logic simulator towards a target property. However, key issues still need to be addressed before this framework can truly deliver on it's promise. Concretizing, or finding a real trace from an abstract trace, remains a hard problem. Abstract traces are often spurious, for which no corresponding real trace exits. This is a direct consequence of the abstraction being an *over-approximation* of the real design. Further, the way in which the abstract model is constructed is an open-ended problem which has a great impact on the performance of the simulator.

In this work, we propose a novel approaches to address these issues. First, we present a genetic algorithm to select sets of state variables directly from the gate-level net-list of the design, which are highly correlated to the target property. The sets of selected variables are used to build the *Partition Navigation Tracks* (PNTs). PNTs capture the behavior of expanded portions of the state space as they related to the target property. Moreover, the computation and storage costs of the PNTs is small, making them scale well to large designs. Our experiments show that we are able to reach many more hard-to-reach states using our

proposed techniques, compared to state-of-the-art methods.

Next, we propose a novel abstraction strengthening technique, where the abstract design is constrained to make it more closely resemble the concrete design. Abstraction strengthening greatly reduces the need to refine the abstract model for hard to reach properties. To achieve this, we efficiently identify sequentially unreachable partial sates in the concrete design via intelligent partitioning, resolution and cube enlargement. Then, these partial states are added as constraints in the abstract model. Our experiments show that the cost to compute these constraints is low and the abstract traces obtained from the strengthened abstract model are far easier to concretize.

To Julia, without your friendship nothing would have been possible.

Acknowledgments

I would like to thank my advisor, Dr. Michael S. Hsiao, for all the great ideas and the wisdom he bestowed to me, which made this work possible.

Ankur Nimish Parikh

Virginia Polytechnic Institute and State University

June 15, 2007

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Design Verification

Growing advances in size and complexity of industrial hardware designs along with decreasing time-to-market requirements have put a lot of onus on verification to ensure that designs are bug-free. Various verification techniques have matured over the years to address the problem of bug detection, each technique having its own strengths and weaknesses. These techniques can broadly be classified as one into one of the following three categories:

- Hardware emulation
- Software simulation
- Formal verification

Hardware emulation uses Field Programmable Gate Arrays (FPGAs) to speed up the verification process by several orders of magnitude. While it can increase the throughput input stimuli application over simulation, its flexibility is limited in monitoring arbitrary properties, especially those temporal ones. In addition, hardware emulators remain expensive, and it remains a time consuming process to map the design to an emulator.

Software simulation is by far the most prevalent method used for design validation today. A netlist of the design is typically simulated with a set of input vectors. The output vectors are captured and checked for correctness. Software simulation is relatively cheap and scalable, and forms the workhorse of verification in industry today. On the down side, simulation can seldom verify corner case properties that are deeply embedded in the design state space.

Formal verification is increasingly gaining importance for hardware design verification. Formal verification refers to a set of techniques that apply mathematical reasoning to validate properties in a hardware design described as a finite state machine (FSM). Such methods are *complete*, i.e., given enough time and space resources, formal methods are guaranteed to prove or falsify a property, and that result would hold for all possible input vectors.

1.2 Hybrid Verification

A general theme that has been successfully applied is one where multiple verification techniques are utilized in such a way that they synergistically complement each other. Such integration has to be carried out in a well though out manner, so that the overall hybrid technique becomes more than the sum of the individual techniques.

Formal verification continues to progress through advances in model checking, symbolic model checking, and bounded model checking, which have greatly expanded the capacity of formal verification tools. Simulation, however, remains the defacto workhorse for industrial hardware verification. Simulation provides an unparalleled ability to scale to ever increasing design size and complexity, primarily because it performs no analysis of the state space of the design. However, it is this very same attribute that forms simulation's greatest drawback: having no clear map of the state space, it is exceedingly difficult for simulation to find bugs in hard-to-reach corner areas of the state space. Model checking, on the other hand, derives considerable information exploring the entire state space of the design, but it comes with a high time and space cost, making the approach viable only for small to medium

sized designs.

Noting the complementary natures of simulation and formal techniques, researchers have long tried to combine the completeness of formal techniques with the speed, capacity and scalability of simulation. Early ideas to such hybrid approaches involved augmenting simulation with small amounts of exhaustive, but bounded formal search on the design. For example, given a target state, a few preimages could be computed to enlarge the target set, which might be easier to hit using random simulation [31, 32, 33]. Alternatively, other approaches have exhaustively explored neighborhoods around heuristically promising states encountered during simulation [13, 21, 12].

The common problem amongst all these approaches is that they perform formal analysis directly on the concrete circuit. This constrains the amount of formal analysis possible, or else there would be an explosion in computational complexity.

1.3 Abstraction Guided Simulation

A complementary approach is to formally analyze a simplified (abstract) version of the design. The abstract design can be simplified enough to be amenable to full formal verification. Such an analysis gives a global view of the structure of the state space of the simplified design, which can be exploited by a simulator to pursue interesting states. Most of the research on abstraction-guided simulation has used abstract preimages from abstract target states as an approximate distance metric to guide the simulator towards concrete target states [25, 23].

Though intuitively appealing, abstraction-guided simulation has its share of problems, chiefly that of dead-end states. Dead-end states are states for which there exists transitions to states closer to the target in the abstract circuit, but no such transition exists in the concrete circuit. This occurs due to the very nature of abstraction: an abstraction of a circuit has more behavior than the original concrete circuit. Since an abstraction-guided search is guided solely by distance metrics obtained from an abstract preimage, it is highly susceptible to get stuck in a local optima dead-end state. Worse, the simulator would have

no indication of whether it is stuck or if it must search harder to make progress.

Consider the figure shown in 1.1. The complete preimage of an abstract circuit is determined, and the simulator maps each concrete state encountered during simulation to one of the corresponding abstract states. Thus, the simulator can bias it's search by preferring those concrete states which have a corresponding abstract state with the smallest distance to the target. However, such a greedy approach gets stuck in local optima, or dead-end states, because the abstraction is an *over-approximation* of the behavior of the concrete design. Many transitions which are possible in the abstract design are impossible in the concrete design, hence for any given concrete state, the simulator has to explore a large breadth of next states in order to find the state which can legally transition to the next closer distance.

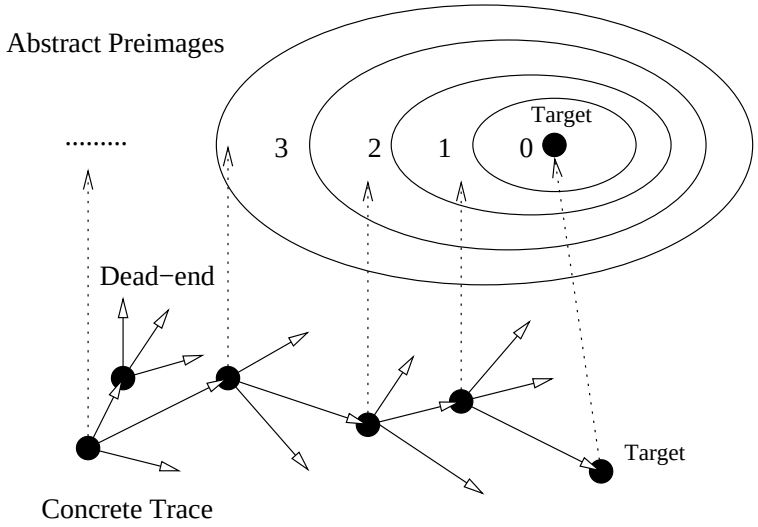

Figure 1.1: Abstraction Guided Simulation

The utility of the abstract preimages depends greatly on the quality of the abstract circuit. For example, an abstraction that captures the behavior of the state space that is unrelated to the target state is of little use. Ideally, the abstraction should capture the behavior of all the variables that can affect the target.

1.4 Previous Work

In the specific-domain of guided logic simulation, one of the first efforts is by Yang *et al.* [32]. This work proposes to direct a random simulator to a target state by enlarging a verification target through backward preimage computation, so that it is sufficient for the simulator to hit any of the states in the enlarged target. However, since the preimage computation is performed on the actual concrete design, it is only practically feasible to go 4 or 5 time steps. Consequently, for hard-to-reach target states, it is difficult for the simulator to reach any of the states in the enlarged target.

In [19], a probabilistic guiding algorithm is presented, which assigns values to design states based on their estimated probability to reach the target. As values are assigned by approximate analysis, there is no apparent mechanism to escape from dead-end situations.

An approach that attempts to reach a target by exploring a range of potential next states in a simulation environment was suggested in [10]. Their solution is a cost function based on the hamming distance between the current configuration reached by logic simulation and the target state. At each step of simulation, a set of alternative next states is considered and the one leading to the minimum hamming distance state is chosen. The advantage is that the computation of the hamming distance can be performed very efficiently at the time of simulation. The downside of this approach, however, is that this measure is usually not a good indication of the distance to the target state and could mislead the simulator as it is possible that two adjacent states in a state transition graph of sequential system have very high hamming distances. A subsequent work in this direction [30] adds the use of automatically-generated "lighthouses", intermediate goals to direct the simulator toward a goal deep in the design.

Amongst hybrid techniques, the recently proposed abstraction-guided simulation [25] is a particularly promising framework which leverages the power of model checking with the low cost and scalability of random simulation. In this approach, an aggressive abstraction of the circuit is first obtained by breaking open many of the state variables in the circuit,

making these state variables primary inputs/outputs of the system. Reachability analysis is performed on the resulting abstract circuit, where the preimage states are obtained and the distances to a target state in the abstract circuit is are computed. Random simulation is then performed on the original (concrete) circuit, using the abstract distances to guide the simulator towards the target state.

The abstract circuit consists of only design modules that closely interact with the property being verified. High-level design information is used to identify such modules. Further, the selection of the abstract state variables is not automated, and requires manual analysis by a skilled user. The chief drawback of this approach is its reliance on the availability of a high-level design specification. Further, the method is not automated and its effectiveness is largely determined by the skill of the user in selecting an effective abstraction. Pure random simulation is employed and next states are picked in a greedy manner, which is highly susceptible to get stuck in local optima (dead-end states).

The authors of [23] employ *localization reduction* to obtain the abstract circuit. Localization reduction relies on the observation that many properties are *local*, that is, they can be verified by considering only variables specified in the property and those in their vicinity. Thus the abstract circuit is obtained by dropping state variables not specified in the target property from the transition relation. This method iteratively *refines* the abstract circuit, if the current abstraction is unable to provide sufficiently detailed preimages for random simulation to reach the target. Here, refining the abstraction means adding more variables to the abstract circuit. The cost of formal analysis on the abstract circuit increases exponentially in the number of abstract circuit variables. In the worst case, the abstract circuit is refined to the point where it is equivalent to the original concrete circuit, which is almost always infeasible.

1.5 Our Contributions

The key drawback of the previous approaches has been their susceptibility to get stuck at dead-end states, and the exponential increase in computational cost incurred in order to get a more refined view of the state space. We address both these issues directly in this work.

We introduce the concept of a *Partitioned Navigation Track* (PNT) (figure 3.1), a collection of abstract preimages built using partitioned sets of state variables. The abstract models built using these partitioned sets of variables are particularly useful because:

1. they represent the abstract behavior of a diverse expanse of the state space
2. the variables are strongly correlated to the target property

Next, we propose a genetic-algorithm based simulation engine that uses fitness function derived from the PNT, and is extremely efficient at generating input vectors that transition the circuit towards the target state. The simulator also provides for means of stepping out of local optima.

Lastly, we propose a preimage strengthening method to avoid spurious states in the abstract preimage. As a pre-processing step, we rapidly identify sets of functionally illegal states of the circuit. We use these unreachable states to constrain the abstract preimage, which results in far fewer spurious states and purely abstract transitions in the abstract preimage.

1.6 Thesis Organization

The remainder of the thesis is organized as follows:

- Chapter 2 introduces preliminary concepts and terms used in the sequel.
- Chapter 3 explains Partitioned Navigation Tracks and the simulation framework.

- Chapter 4 describes the pre-processing step to identify illegal states in a circuit, and its application to strengthen the PNTs.
- Chapter 5 concludes the thesis.

Chapter 2

Background

2.1 Iterative Logic Array

Figure 2.1 shows the model of a synchronous sequential circuit. It consists of a set of primary input signals, a set of primary output signals, a block of combinational logic, and a set of flip-flops or state variables. The next state values for the flip-flops are determined by the logic realized by the combinational block together with the present state values of the flip-flops and the current primary input vector.

The iterative logic array (ILA) of a synchronous sequential circuit is constructed by connecting together multiple copies of the combinational logic block of the circuit. The next

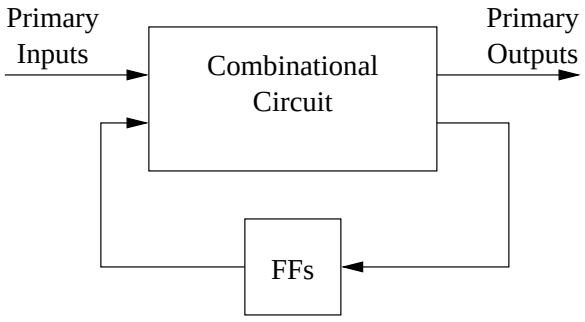

Figure 2.1: Synchronous Sequential Circuit

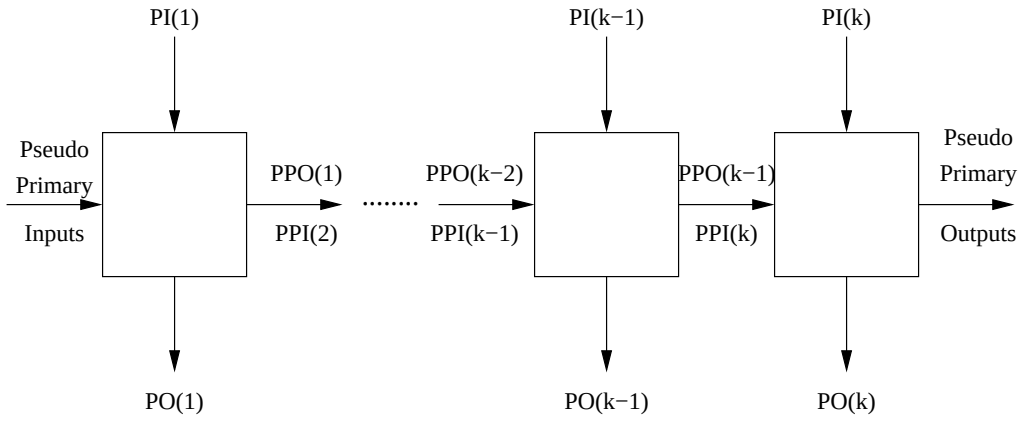

Figure 2.2: Iterative Logic Array

state variables of each cell of the iterative logic array are connected to the present state variables of the cell following it. An iterative logic array of a sequential circuit expanded for k timeframes is shown in Figure 2.2. The primary inputs for all cells in the ILA are controllable and the primary outputs for all cells are observable. Furthermore, the present state variables of the left-most cell (cell 1) are also fully controllable, and the next state variables of the right-most cell (cell k) are observable [1, 24].

2.2 Boolean Satisfiability

Boolean satisfiability is the problem of determining if the variables in a Boolean formula can be assigned values in such a way so as to make the entire formula evaluate to TRUE. If such a variable assignment exists, the Boolean formula is termed as *satisfiable* (SAT). If no combination of Boolean value assignments to the variables of the formula can make it evaluate to TRUE, then the formula is termed *unsatisfiable* (UNSAT).

2.2.1 Conjunctive Normal Form

For the satisfiability problem, a Boolean formula is usually represented in the *conjunctive normal form* (CNF), in which the formula is a conjunction of *clauses*, where each clause is a disjunction of *literals*. A literal is a Boolean variable, or its negation.

Every arbitrary Boolean formula can be converted into an equivalent formula that is in CNF form by applying DeMorgan's law's, the distributive law, and the double negative law in Boolean logic. Formulae in CNF form are generally preferred for determining satisfiability, as the formula would be satisfiable if and only if every clause in the formula can be evaluated to TRUE.

2.3 Model Checking

Model checking [7] is an automatic formal verification technique, in which the design is modeled as a finite state machine. Specifications are expressed as *temporal properties*, which define the behavior of the system with respect to time. For example, the property $F(p = 1)$ indicates that *eventually*, the signal p is asserted in the system. Alternately, the property $G(q = 0)$ indicates that the signal q is *always* low. The reachable states of the finite state machine are traversed to verify the properties. In case a state is reached which violates the property, a counterexample, which consists of a sequence of states, is generated [5, 16]. In general, properties are classified as *safety* and *liveness* properties. While safety properties define what should *not* happen, liveness declare events that should eventually take place in the system.

A key bottleneck in model checking is the size of the finite state machine that can be handled. Since the number of states in an FSM grows exponentially with the number of state variables, it quickly becomes infeasible to enumerate all states even for medium-sized designs. In *symbolic model checking*, the states of the FSM are represented implicitly using Boolean functions. For example, in a system with two state variables, say v_1 and v_2,

which can take the values (11, 10, 01), then the state space can be represented using the Boolean formula $v_1 \vee v_2$, instead of explicitly enumerating the states. Reduced ordered binary decision diagrams (ROBDDs) [4] provide an efficient, canonical data-structure to represent and manipulate Boolean functions, and have greatly increased the capacity of symbolic model checking tools to deal with larger designs. ROBDDs, however, are sensitive to the variable order used to represent Boolean the function. A non-optimum variable order can result in the data-structure size to explode for large, complex functions. Further, determining the optimum variable order is NP-complete; the time needed to determine the optimum variable order can be exponential in the size of the Boolean formula.

To overcome size explosion of BDD's when dealing with large functions, researchers have exploited advances in SAT solving techniques [26] to further enhance the capacity of model checkers. *Bounded model checking* was proposed in [3], in which the model checking problem is converted to a Boolean satisfiability problem, and properties verified over a finite *bounded* number of steps.

2.3.1 Representing Sequential Circuits

A sequential circuit can mathematically be represented by a finite state machine (FSM), which is a 6-tuple $(S, I, O, \delta, \lambda, s_0)$ where S is the set of states, I is the set of input values, O is the set of output values, $\delta : S \times I \rightarrow S$ is the next state function, $\lambda : S \times I \rightarrow O$ is the output function, and $s_0 \in S$ is the initial state.

2.3.2 Image and Preimage

The *image* of a set of states, A, is the set of states, B, that can be reached from A by applying any input vector. In other words, if a sequential circuit is in some state $s \in S$, then the image of s is the set of all possible next states that the circuit can transition to. Formally, the image of s is defined as:

$$Image(\delta, s) = \{s' \mid \exists i \in I, s' = \delta(s', i)\}$$

The image operation can be iteratively computed until no more reachable states can be added. At this point, a fixed point has been reached. In practice, generally the image of the reset state of the design is computed until the next image set contains the target state of interest, as shown in figure 2.3.

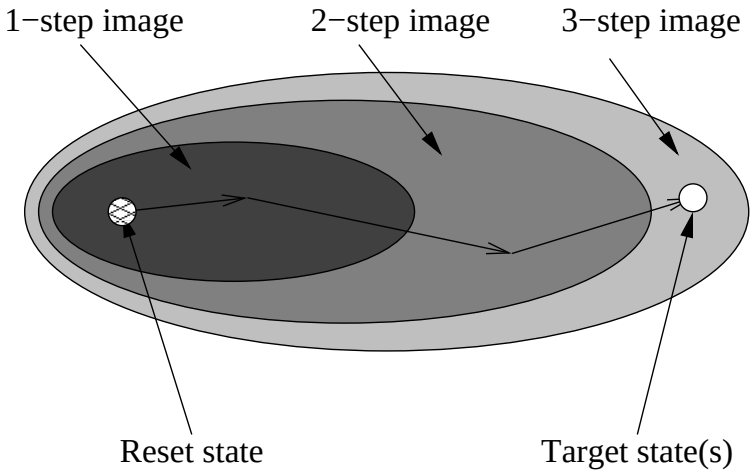

Figure 2.3: Forward Image Computation

Conversely, the *preimage* of a state $s' \in S$ is defined as the set of states from which the sequential circuit can transition to state s' by application of an input vector. Thus, if the sequential circuit is in some state s, and there exists an input vector whose application would result in the next state to be s', then s belongs to the preimage of s'. The complete preimage set consists of all such states s. Formally, the preimage of s' is defined as:

$$Preimage(\delta, s') = \{s \mid \exists i \in I, s = \delta(s', i)\}$$

2.3.3 Fixed Point Image and Preimage

Computation of the image or the preimage of a state to a *fixed point* means iterating the image or preimage computations until the set of states obtained in two consecutive iterations remain the same. Figure 2.4 illustrates this for a preimage computation of a target state.

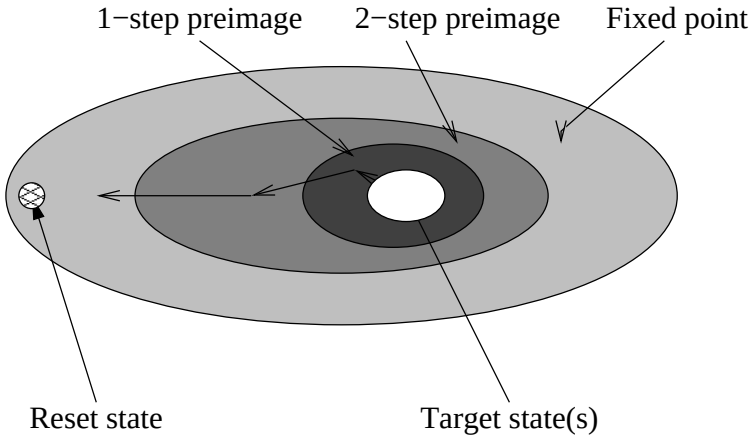

Figure 2.4: Backward Preimage Computation to a Fixed Point

Let $T_0 \subset S$ be a set of initial state(s), then computation of the i^{th}-step image of T_0 under δ is defined as follows:

$$T_i = Image(\delta, T_{i-1}), 1 \leq i$$

A fixed-point is reached whenever $T_i = T_{i-1}$, because when condition holds, the image computation will yield no new states.

Similarly, the i^{th}-step preimage of some target state(s) $Q_0 \subset S$ is defined as:

$$Q_i = Preimage(\delta, Q_{i-1}), 1 \leq i$$

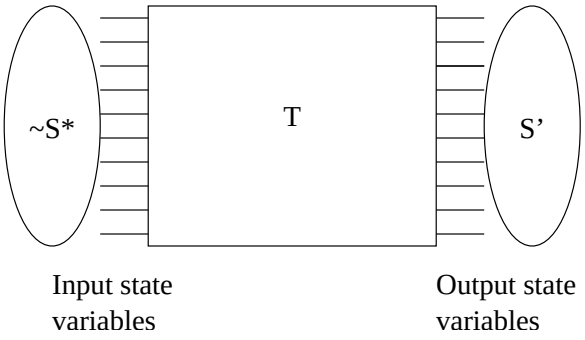

Figure 2.5: One-timeframe ILA to compute preimage

2.3.4 Using satisfiability to compute preimages

In this work, we extensively employ preimage computation, and we use a SAT solver to compute preimages of states to a fixed point. The following describes the method used to compute preimages in this work.

Let T be the transition relation of the design, I be the set of initial state variables, O be the set of output state variables, S' the target state set whose preimage is to be computed, and $S*$ be the set of preimage solutions computed so far. The combinational portion of the design (i.e., the transition relation T_1) is first converted to a CNF formula. Then, a satisfying solution to the following CNF formula:

$$(I \wedge \neg S*) \wedge T_1 \wedge (O \equiv S')$$

yields a new preimage state (not contained in $S*$) of the target state set S'. To find the entire preimage set, the new solution is added to $S*$, and the procedure is repeated until the resulting instance in unsatisfiable. We term the CNF expression $\neg S*$ as a *blocking clause*.

Figure 2.5 pictorially shows the SAT instance as a one-timeframe ILA with the constraints on the input and output state variables.

It should be noted that the preimage set computed using this method is in fact an

over-approximation of the true preimage set. While the final set does contain all the states in the true preimage, it also contains additional states that are unreachable in the design.

2.4 Circuit Abstraction

Circuit abstraction attempts to reduce the space complexity of the design state space, while at the same time retaining enough information to give vital clues as to the essential characteristics of the state space.

2.4.1 Localization reduction

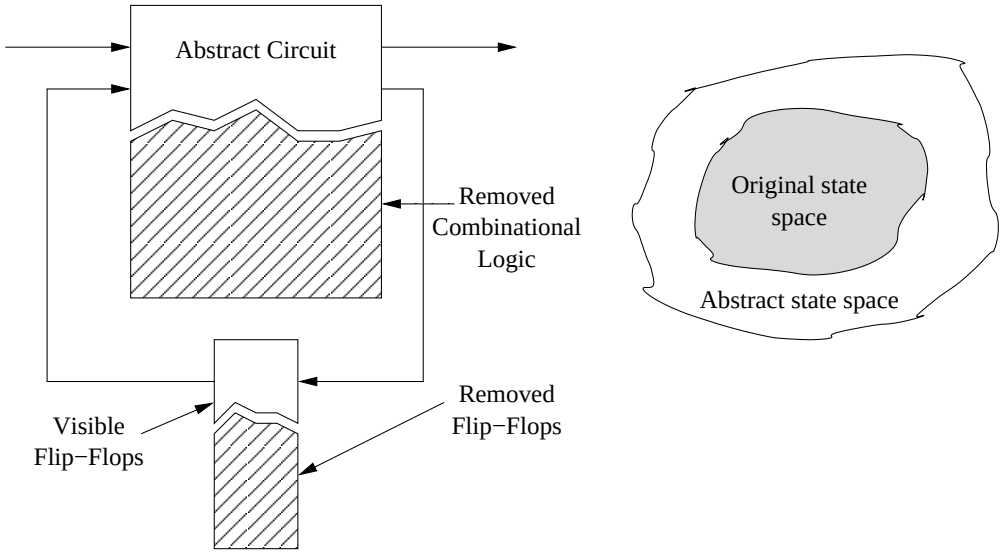

Figure 2.6: Localization Reduction

A commonly used abstraction technique for sequential circuits is *localization reduction* [20, 28, 6], illustrated in Figure 2.6. The abstract model is created from the concrete circuit by removing a large number of flip-flops together with the logic required to compute their next state. The flip-flops removed are called *invisible*. The flip-flops remaining in the abstract

model are termed *visible*. The initial abstract model is created by making the flip-flops present in the target property visible, and the rest invisible.

Localization reduction results in a abstract circuit which is an over-approximation of the original circuit for reachable properties, because the constraints that were present in the original circuit due to the invisible flip-flops are no longer present in the abstraction. Thus, if a target state is shown to be unreachable in the abstract circuit, we can safely conclude that it is definitely unreachable in the original circuit. The reverse, however, is not true. The drawback of abstraction is that when a target state is shown to be reachable in the abstract circuit, the abstract trace produced might not correspond to any concrete trace in the original circuit. In order to check if an abstract trace is spurious, it has to be simulated on the concrete machine, a process known as *concretization*. A popular approach translates the concretization problem into propositional satisfiability. The full concrete transition relation is unrolled as many times as there are transitions in the abstract trace to be checked, and the SAT solver checks if there is a path in the resulting circuit that is consistent with the values of the visible variables in the abstract trace. If the instance is unsatisfiable, the abstract trace is spurious, and *abstraction refinement* has to be performed.

2.4.2 Abstraction Refinement

The basic idea of abstraction refinement [20, 8, 9] is to create a new abstract circuit which more closely resembles the original circuit (i.e., more visible flip-flops, as shown in Figure 2.7) in order to prevent the spurious trace. In an abstraction refinement approach, concretization and refinement steps are iterated until the target state has been reached or proved unreachable. A major drawback of this approach is that which each concretization step, the complexity of generating a new abstract trace increases exponentially.

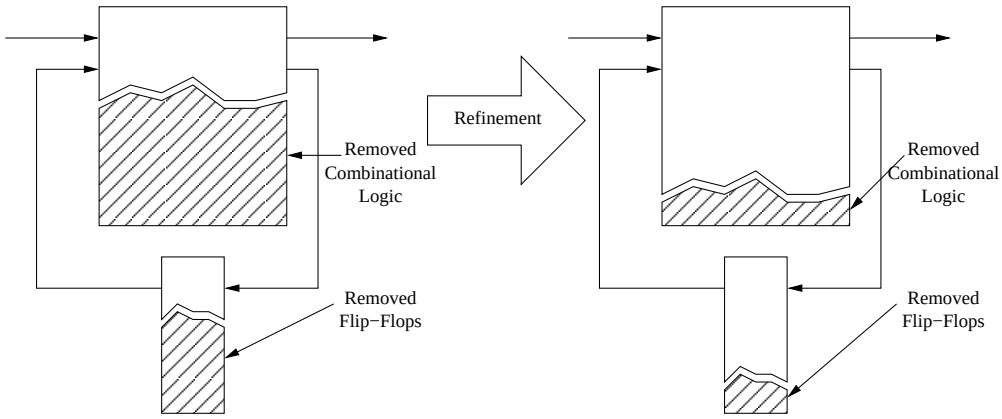

Figure 2.7: Abstraction Refinement

2.5 Guideposts

Guideposts[32] are designer provided *hints* to guide the search towards the target. The hints, called Guideposts, are a series of conditions that the designer believes to be interesting, or even required preconditions, for the circuit to transition to the target state. A simulator can get a good indication of how far away it is from the target state by keeping track of the number of guideposts that the current state and its ancestors have past through. Thus a simulator can discriminate between candidate starting states and pursue the more promising states.

2.6 Genetic Algorithms

Genetic Algorithms (GAs) [14, 11] belong to the class of evolutionary computing. Problems are solved by an evolutionary process resulting in a best (most fit) solution (survivor). The following notations are commonly used in the context of genetic algorithms:

- *Individual/population:* A candidate solution for the problem at hand
- *Fitness:* Each individual is associated with a fitness value, which measures the quality of this individual for solving the problem.

- *Selection/crossover/mutation:* Typical GA operations to reproduce a new population from an existing population.

- *Generation:* When a new population is reproduced from a previous population via GA operations.

A population of individuals evolves over a number of generations using the GA operations of selection, crossover, and mutation, and the average fitness of one generation is expected to improve over previous generations. The evolutionary process ends when the desired target is achieved or a preset amount of resources have been exhausted.

2.6.1 Selection operator

Selection is the stage of a genetic algorithm in which individuals are chosen from a population for breeding the next generation. While there are several generic selection algorithms, the one used in this work is *fitness proportionate selection*. In fitness proportionate selection, the fitness value associated with an individual determines its probability of selection. While candidates with high fitness are most likely to be selected, there is a small chance they may be eliminated and some weaker solution survives. This is advantageous, as even though a solution may be weak, it might contain some component which could prove useful in the recombination process. Figure 2.8 illustrates the selection process.

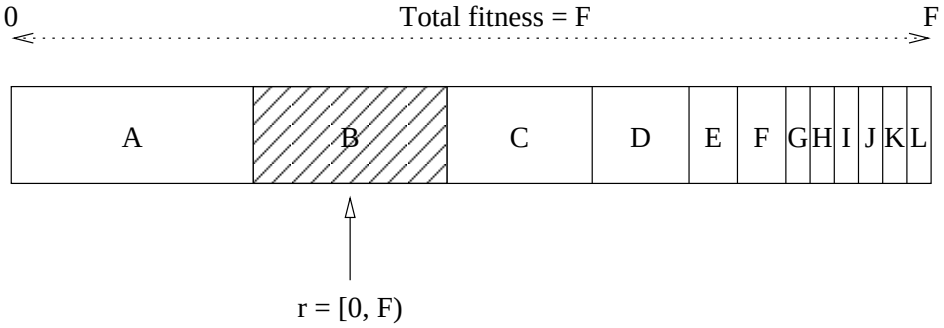

Figure 2.8: Fitness Proportionate Selection

2.6.2 Crossover operator

The crossover operator is used to create the next generation using the selected individuals from the current generation. While many crossover approaches exists, in this work we employ the *cut and splice* technique, illustrated in Figure 2.9. A crossover point is identified in both the parents, and each parent individual is cut at that point. Pieces from each parent are spliced together to get two new children. It must be noted that the children length of the children might differ from the parents.

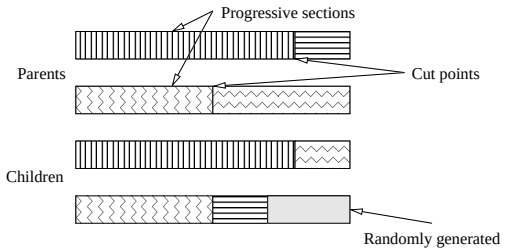

Figure 2.9: One-point crossover

Chapter 3

Partitioned Navigation Tracks

3.1 Motivation

Though intuitively appealing, abstraction-guided simulation has its share of problems, chiefly that of dead-end states: states for which there exist transitions at a closer distance to the target in the abstract circuit, but no such transition exists in the concrete circuit. This is because an abstraction of a circuit has more behavior than the concrete circuit. Thus, a state that might appear to be one transition away from the target in the abstract circuit is likely to be much further away in the original circuit. Since the search is guided solely by the distance values obtained from the abstract circuit, it has no way of knowing whether it is stuck at a dead-end,or whether it much search harder to make progress. Further, the utility of the approximate distance information obtained from an abstract circuit depends greatly on the usefulness of the visible state variables selected. For example, an abstraction using state variables that are unrelated to the target would not yield much useful distance information. Ideally, the abstraction should capture the behavior of all the variables that can influence the target state.

Figure 3.1 illustrates the motivation behind our approach with a simple example. We first divide the state variables into partition sets. Then we introduce the concept of *Partition*

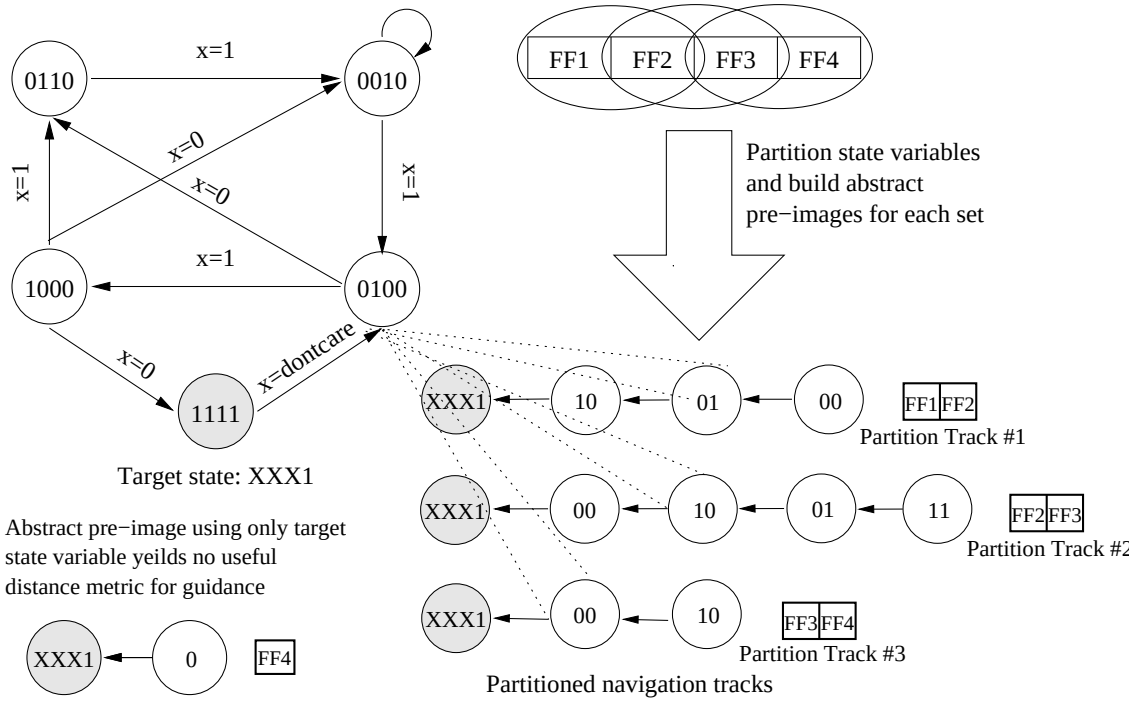

Figure 3.1: Partitioning state variables to form multiple abstractions

Navigation Tracks (PNTs), a collection of abstract preimages built on the partition sets. Each preimage denotes a set of abstract states from which a path exists to the target. The variables in these partition sets are particularly useful because:

1. they are highly correlated to the target property.

2. they represent the abstract behavior of a diverse expanse of the state space.

Both of these facts contribute to the PNTs providing our search with approximate distance metrics that are far more accurate than a single abstract preimage as proposed in [25]. Further, as the size of the partition sets is limited, the cost associated with computing the partition navigation track increase only linearly with each additional partition. Thus we have the freedom to increase the number of partition states used without incurring an exponential increase in the cost of formal analysis, as in the case with abstraction refinement techniques proposed in [23]. In [29], while multiple partition state sets are used to guide the search, no preimage states are computed to capture the reachability knowledge within each partition.

Other previous works in this area have usually coupled a guiding mechanism with pure random simulation. We reckon that a more intelligent search strategy would yield far better results. We implement a search engine based on genetic algorithms (GA) [14, 11]. Our approach is particularly effective as we intelligently seed the GA with abstract primary input (PI) vectors obtained during PNT creation. Experiments show that our technique is very effective in reaching hard-to-reach states where previous methods fail.

3.2 The Proposed Approach

3.2.1 Overall Framework

The basic flow of our approach is shown in Figure 3.2. In the abstraction engine, partitioned sets of state variables are created using state variables specified in the target property, along

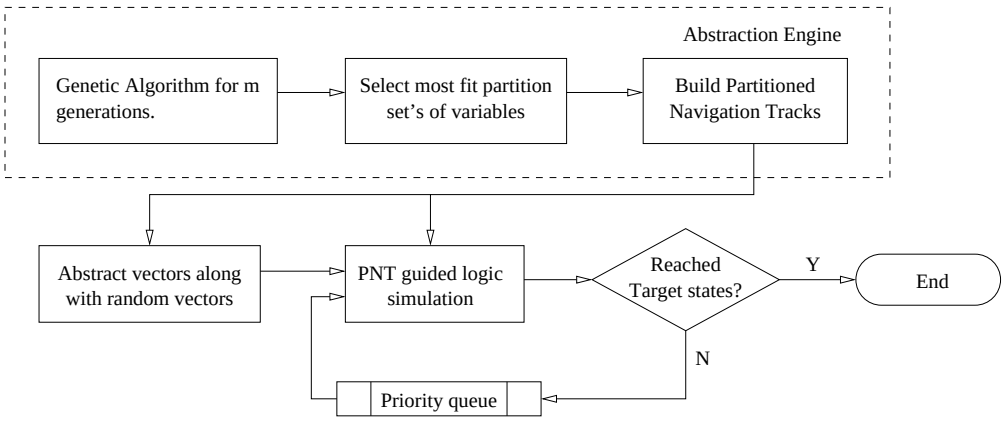

Figure 3.2: Overall Verification Framework

with other variables that are deemed to be highly correlated to the target property variables. Next, partitioned navigation tracks (PNTs) are computed for these partition state sets using a satisfiability (SAT) engine. Once the PNTs have been computed, they are used to guide a GA based logic simulator to the target state. In the concrete circuit, starting from the initial state, logic simulation is guided in the following manner:

- Primary input vectors generated by the SAT engine while computing the abstract preimages for the PNTs are collected, and along with random vectors they form an initial population of simulation vectors.
- A priority queue stores interesting states visited during logic simulation. The queue orders states by their estimated distances to the target.
- A starting state is chosen from the head of the queue, and the GA evolves the input vector population for m generations. Interesting states visited during simulation are added to the priority queue.
- A shallow, bounded SAT instance is attempted if the simulation is about to timeout for lack of progress, otherwise the next best state from the head of the priority queue is chosen and simulation is pursued from there on.

3.2.2 Abstraction of State Variables

The quality of variables selected to compute the abstract preimages plays a critical role in determining the effectiveness of the distance information provided to the simulator. A good abstraction precisely captures the necessary conditions required for the design to transition to the target state.

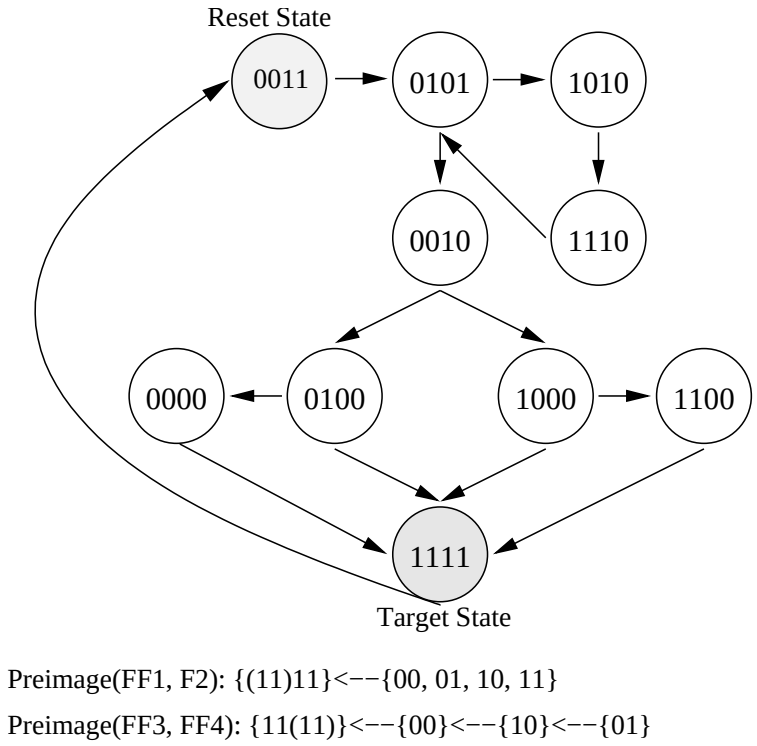

Figure 3.3: Selecting Variables for Abstraction

Consider the simple FSM illustrated in Figure 3.3. The machine has four state variables, $FF1, FF2, FF3, FF4$, and the target state is 1111. Suppose $(FF1, FF2)$ are selected as our abstraction variables. In this case, the abstract preimages yield no useful information, as it indicates that every state is one transition away from the target. However, if variables $(FF3, FF4)$ are selected, the abstract preimage captures the precise necessary conditions required on variables $FF3$ and $FF4$ for the design to transition to the target state. A simulator can thus use this preimage to guide its search towards the target, by preferring those

concrete states that satisfy the necessary conditions on $FF3$ and $FF4$. Further, we want to select *multiple* partitioned sets of state variables to capture the behavior of different portions of the state space. Since several concrete states might map to a single abstract state, we can effectively differentiate between such states by comparing them against multiple abstract preimages.

We employ a simple genetic algorithm (GA) [11] in order to evolve fit partitioned sets of abstraction variables. A random initial population $I = \{I_1, I_2, ..., I_n\}$ of state sets is created. Each state set consists typically of 8 to 10 randomly chosen state variables.

The abstract preimage of each individual in the population is computed to a fixed point. Note that since each state contains between 8 to 10 variables, the complete preimage set for any individual can have at most 256 to 1024 states, which is manageable. The fitness value is determined by the number of distinct preimage state sets in the complete preimage. For example, if $\{A, B, C, D, E, F\}$ are abstract states in a design, F being the target state, and the abstract preimages of two different individuals I_1 and I_2 are found to be:

$$I_1 = \{A, B, C\} \rightarrow \{D, E\}, \rightarrow \{F\}$$

$$I_2 = \{A, B\} \rightarrow \{C\} \rightarrow \{D\} \rightarrow \{E\} \rightarrow \{F\}$$

Clearly, I_2 is the better fit of the two individuals. It indicates that the states C, D, and E must be traversed in that order for the design to be in state F. The preimage of partition I_1, on the other hand, provides a more ambiguous view of the state space. It suggests that either states D or E can transition to F. Should the simulator rely on the abstract preimage I_1, it might get stuck pursuing *dead-end* state D, which has no real transition to F. Numerically, $fitness(I_1) = 2$ and $fitness(I_2) = 4$, which are the number of distinct preimage sets in each partition.

After a fitness value has been assigned to all individuals, a set of individuals are selected for breeding the next generation. The best fit n individuals are always selected. An

additional $m < n$ individuals are selected using fitness proportionate selection. While the best fit of the remaining individuals have the best chance of being selected, a small number of less fit individuals are also selected in this selection scheme. This ensures population diversity for the next generation, so that the population of abstract variables does not get limited to a localized corner of the state space.

The set of selected individuals is split into two halves, one half containing the best-fit half of the individuals, the other containing the least fit individuals. Two child individuals in the next generation are created by performing a *one-point crossover* between two randomly selected parent individuals from each half of the current generation. The crossover point is randomly selected, and each child individual inherits portions of both parents, according to the crossover point.

The final partitioned state variable set is obtained after g generations. The preimages of the best fit n individuals of the last generation form the final partitioned navigation tracks.

3.2.3 Computing the Partitioned Navigation Tracks

Once we have selected the abstraction variables, we compute the PNTs using a satisfiability (SAT) engine. Given that the selected set of abstraction variables is $A = \{A_1, A_2, ..., A_n\}$, the abstract preimage for each partitioned set A_j is computed to a fixed point. Since we only want *abstract preimages*, the complete satisfying solution returned is projected onto the abstraction variable set A_j. The only solution values considered are those of the the variables present in A_j, and the rest of the variables are considered as unspecified. The i^{th}-step preimage for partition A_j is computed as follows: (i) The concrete design is unrolled for one-timeframe and converted to a CNF formula. (ii) Constraints are added to this formula as shown in the following expression:

$$(I \wedge \neg S_j * \wedge \hat{P}^i_{j-1}) \wedge T_1 \wedge (O \equiv S')$$

where I is the initial state variables, $\neg S_j*$ is the blocking-clause that prevents repeated computation of solutions already found for the current preimage, T_1 is a one-timeframe transition relation for the circuit, O is the output state variables, and S' is the set of target states. $\hat{P}_j^i$ further constrains the input state variables by constraining variables that appeared in abstraction sets $\{A_1, A_2, ..., A_{j-1}\}$ to values contained in up until their respective i^{th}-step preimages, which were previously computed. In effect, the $\hat{P}_j^i$ clause tightens the abstract space by precluding those states whose preimages conflict with the abstract preimages computed so far. $\hat{P}_j^i$ is defined as

$$\hat{P}_j^i = \begin{cases} \bigwedge_{x=1}^{j} \bigvee_{y=1}^{min(l,i)} P_{x,y} & \text{if } j > 1; \\ \phi & \text{if } j = 1. \end{cases}$$

where l is the number of distinct preimage state sets in preimage P_x (which is the abstract preimage computed using abstraction variables in set A_x), and $P_{x,y}$ is the set of states contained in the y^{th}-step preimage of P_x.

The output state variables are constrained to S', which is defined as:

$$S' = \begin{cases} TargetProperty & \text{if } i = 1; \\ P_{j,i-1} & \text{if } i > 1. \end{cases}$$

In essence, S' constrains the output state variables to the target property, for the first-step preimage computation for every set of abstraction variables. For all subsequent preimages, the output variables are constrained to the previously computed preimage set.

Let $I(A_j)$ be defined as the subset of input state variables that appear in set A_j. After a satisfying solution to the CNF instance is found, $I(A_j)$ is added to the blocking-clause set S_j*. State variables *not* in A_j are ignored. Since the number of variables in A_j is small, this makes it feasible to compute all solutions to a fixed point for each abstraction set A_j.

3.3 Guided Logic Simulation

3.3.1 Overview

In order to reach the desired target state, we guide our search by pursuing intermediate states that are determined to have the closest abstract distance to the target state, as calculated from the PNTs. Our guided logic simulator is based on a GA engine and the fitness values for each individual is computed as the sum of the abstract distance values obtained from each partitioned track. Algorithm 1 shows an overview of our simulation framework.

Algorithm 1 GA based logic simulation

```
population = abstract PNT vectors + random vectors
push reset / synchronized state to the priority queue
while !target_reached and !timeout do
  start_state = top of priority queue
  for n generations do
    for each individual in the population do
      simulate individual and record each visited state
      if dist(visited_state) ≤ dist(start_state) then
        push visited_state to the priority queue
      end if
      calculate fitness value of this individual
    end for
    perform cross-over/selection to get next generation
    if no progress for ≥ threshold iterations then
      invoke a bounded SAT instance
    end if
  end for
end while
```

An *individual* consists of a sequence of n primary input vectors. We seed the initial population of the GA with the abstract primary input vectors obtained while computing the abstract preimages. Since these vectors traverse different abstract circuits towards the target, we believe they are particularly well suited to traverse the real circuit to the target after genetic evolution. In addition to these vectors, we generate an additional 20% of the initial population randomly.

Since an individual is a sequence of n vectors, the circuit traverses through at most n states when any given individual is simulated. The fitness of the individual is calculated on the basis of the *best* of these n states, i.e., the state that is deemed to have the least abstract distance to the target. The single best state encountered during simulation for every individual is inserted into the priority queue, as long as

$$distance_{best} \leq distance_{start}$$

where $distance_{best}$ is the estimated abstract distance of the best state encountered, and $distance_{start}$ is the abstract distance of the starting state from which the individual was simulated.

Meaningful comparison of the fitness of two individuals can only be made if they were simulated from a common starting state. Thus, when evolving a population, the starting state is kept the same. During the evolutionary process, the priority queue is fill with the best states encountered for each individual. In a greedy manner, the reachable state at the head of the queue is selected as the starting state for the next round of evolution.

Even though the approximate abstract distances obtained from the PNTs are more accurate than a single abstract preimage, there is still a chance that the simulator may be stuck at a dead-end state. Our simulator accounts for such situations by limiting the effort spent pursuing each state. Each starting state is pursued only for g generations, following which it is discarded and the next best state in the priority queue is chosen. In the event that a dead-end state is chosen, no progress would be made, and the state would be discarded after g generations. This provides the simulator with a mechanism to step out of local optima.

On the other hand, if the simulation fails to make progress for *threshold* consecutive starting states, it is possible that the path leading to states at closer distances is narrow, i.e., only very specific primary input vectors can transition the circuit to the closer state. At such a point, it would be very difficult for simulation to generate the precise vectors to

traverse the extremely narrow path. In such instances, we invoke a SAT solver and setup a bounded model checking (BMC) [3] instance to attempt to find a sequence of primary input vectors that would bridge the current state to either the target state or a state with a closer estimated abstract distance to the target. If the BMC instance is satisfiable, the solution is retrieved and the resulting closer state is added to the priority queue. On the other hand, if the instance is unsatisfiable in the specified bounds, the starting state is dropped and we continue with the next state at the head of the priority queue.

3.3.2 Fitness of Individuals

The fitness value for an individual is determined by the closest state reached when the sequence of vectors is logic simulated. The distance of a concrete state encountered during simulation is determined by mapping it to abstract states in the PNTs. An abstract state which covers the concrete state is determined for each track in the PNT. The fitness value is essentially the sum of distances of each abstract covering state. Figure 3.4 illustrates an example.

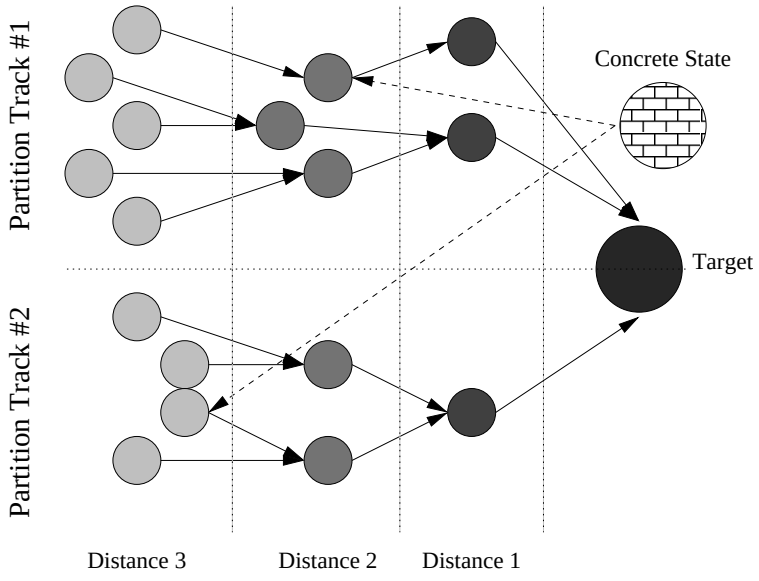

Figure 3.4: Estimating distance to target for a concrete state using PNT

Thus, the best state will be the one with the least estimated abstract distance. Since it is intuitive to have a larger numerical value for fitter individuals, the distance value calculated is subtracted from the largest distance possible. This gives the fitness numerical value, as shown in the following equation:

$$fit_s = \sum_{i=1}^{n} (max_i - d_i)$$

where n is the number of abstract preimage tracks in the PNT, max_i is the maximum distance value of the i^{th} track, and d_i is the distance at which the state s mapped on the i^{th} track. In the case where multiple states evaluate to be equally fit, one of them is randomly selected to be the best state.

3.3.3 Selection and Crossover Operators

To select individuals that will parent the next generation, a fitness proportionate selection procedure is employed. Individuals are randomly selected from the current generation, with more fit individuals have the greater probability of being selected.

An individual may contain certain vectors that can direct the simulator away from the target. Thus, for each individual, we may cut it at the first point at which it pulls the simulator in a direction away from the target. This divides the individual into two sections: (1) a progressive section and (2) a impeding section. We perform the crossover operation by splicing the progressive section with the impeding section of a second individual. The resulting spliced sequence of vectors is the new individual. If needed, the new individual is either truncated or randomly padded to the fixed individual length. In the event that no such cut point can be found, the center point is chosen by default. Figure 3.5 illustrates the crossover method.

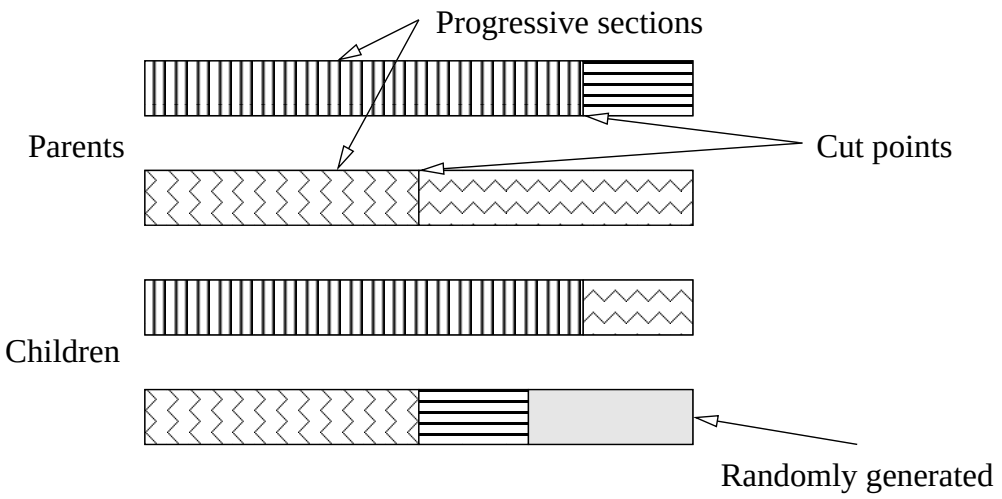

Figure 3.5: Crossover operator

3.3.4 Traversing narrow paths using bounded SAT

Certain hard-to-reach regions of the state space may have few, or sometimes unique, vectors that can lead to the target. It is highly unlikely that simulation can precisely generate the required vectors to navigate such constricted spaces. In the case where the simulator seems to be stuck without making progress for a certain *threshold* consecutive iterations, we invoke a SAT solver to attempt to bridge to a closer state. To keep a check on the computational cost, we create a SAT instance limited to a *bound* as in BMC [3]. If the BMC fails, we abort the current state and pursue others waiting in the queue. In our experiments, the *bound* value is set to the length of an individual. If a solution is found, the BMC would return a witness to the target state (or a state with a closer abstract distance), and we add it to the priority queue. Further, the primary input vectors obtained are added to the population as a new individual for the next generation.

Figure 3.6 shows a typical SAT-based BMC instance. The circuit is unrolled for *bound* number of timeframes, and the initial state variables are constrained to the current starting state under consideration by the simulator. In essence, the BMC attempts to find a sequence of inputs that can bridge the starting state with any of the target states within the given

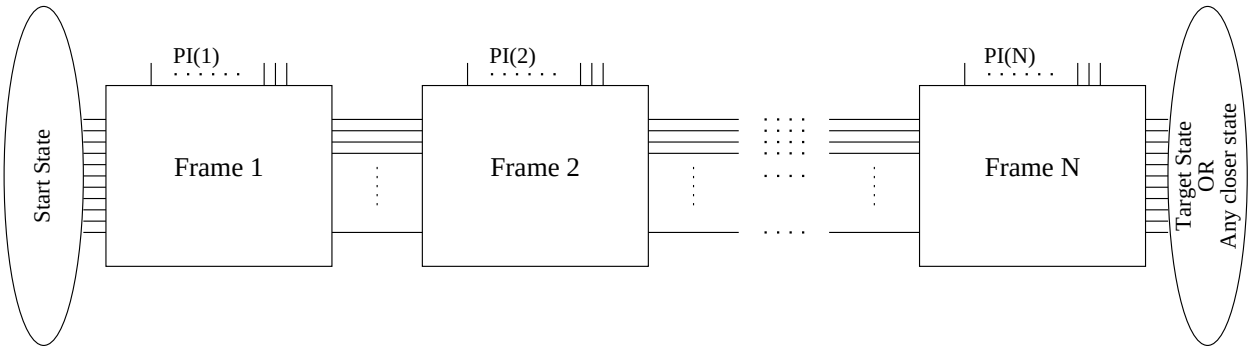

Figure 3.6: Bounded SAT instance to traverse narrow corridors of the state space

bound. The SAT solver is asked to find a solution leading to either the target state, or a state with an estimated abstract distance that is closer than the current starting state. The state variables in the final timeframe are constrained according to the following expression

$$TargetState \vee [\bigwedge_{i=1}^{n}(\bigvee_{j=d_i-1}^{1} States_{i,j})]$$

where n is the number of tracks in the PNT, d_i is the distance at which the current starting state maps to at the i^{th} track, and $States_{i,j}$ is the set of abstract states contained in the i_{th} track at the j_{th} distance.

3.4 Experimental Results

We implemented the proposed approach in C++. Experiments were conducted on ISCAS89 [17] and ITC99 [18] benchmark circuits on a 3.2 GHz Pentium 4 with 1G RAM. The target states used are states aborted by STRATEGATE [15]. An aborted state is a state that STRATEGATE failed to justify within the alloted resource limits.

We compare the performance of our guided simulator when the number of partition sets are used are varied. In a single abstract partition, the states variables are directly

selected from the specified variables in the target property, whereas the k partition sets are computed with our genetic algorithm method. To make the PNTs computation lightweight, in all partition sets, the maximum number of state variables is limited to 8.

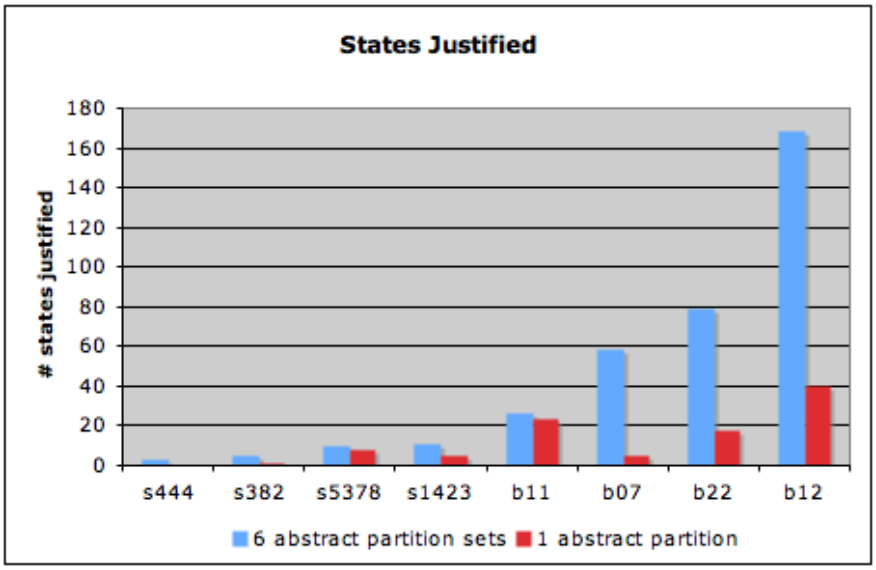

Figure 3.7: States justified w.r.t. number of partitions

Figure 3.7 shows the total number of states that could be justified using 1 partition set and 6 partition sets. For small circuits (s382, s444), there were only a few aborted states available. We can see that as the number of partition sets increased, the number of aborted states that could be reached is greatly increased. This is due to the reason that increasing the number of partition sets increases the accuracy of fitness value calculation, which could more effectively guide the simulator to justify the target states.

Figure 3.8 shows the runtime comparison for 1 and 6 partition sets. While increasing to 6 partition sets doubled the total run time, we will see that this increase is worth the effort when compared with other techniques, described later in this section.

Figure 3.9 shows the runtime and number of states justified for a particular circuit, b07. We can see that the number of states justified can be significantly increased as the number of partition sets increased, without a heavy toll on the computational cost.

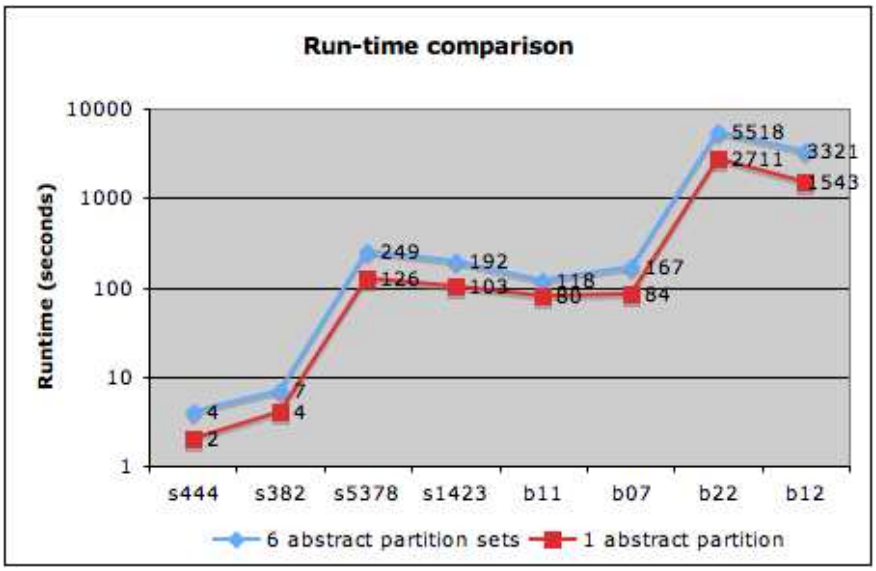

Figure 3.8: Runtime w.r.t. number of partitions

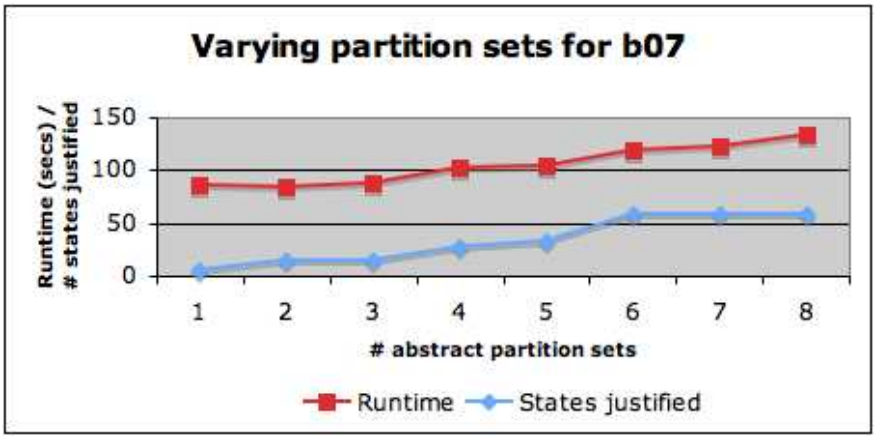

Figure 3.9: Varying partition sets for b07

To demonstrate the effectiveness of our genetic algorithm strategy to pick abstraction variables, we also performed a study to compare our results with a randomly partitioned state sets. Figure 3.10 compares the number of properties verified using GA generated partition sets and random partition sets. To make the comparison fair, the random partition sets are chosen such that the number of state variables in each set and the number of partition sets are the same as the corresponding GA partition sets. As shown in the figure, the GA partition sets are much more efficient than the random sets, which demonstrates that the GA partition sets correlate the state variables in each partition to the target states, in order to guided the search more efficiently.

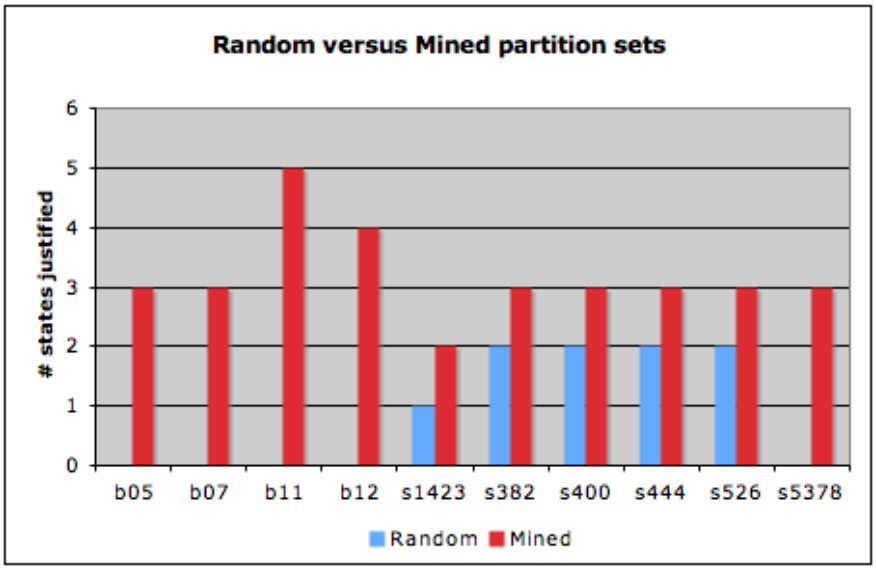

Figure 3.10: Random v/s GA-generated abstract variable partitions

Chapter 4

Abstract Preimage Strengthening

Concretization is the process of constructing a trace in the real machine, given an abstract counterexample. Concretization is the hardest and most time-consuming step in an abstraction guided verification. The difficulty arises due to the very nature of abstraction. The abstract model has "more behaviors" than the concrete design; thus, what might appear to be one transition in the abstract model, might, in reality, take several transitions to achieve in the concrete design. Furthermore, some transitions in the abstract model may not be feasible at all in the concrete design. One way to overcome this problem is via refinement of the abstract model, in which more details (in particular, the state variables) are added back to the abstract model. However, such refinement schemes incur an exponential increase in the computational complexity with the number of added state variables in each refinement. Instead of refining the abstract model, we attempt to *strengthen* the abstract model without restoring any extra state variables. As a result, the cost of formal analysis remains at the same level of complexity, but the resulting analysis provides much more accurate guidance to the simulation engine.

4.1 Motivation

The motivation behind our approach stems from the observation that the difficulty in concretizing an abstract trace arises from the fact that the abstract model often transitions in ways that would be illegal in the concrete design. For example, a one-step transition from any of the state $\{A, B, C, D, E\} \rightarrow \{F\}$ might be possible in the abstract model, but the path in the concrete design might necessarily require transitioning through states $\{A\} \rightarrow \{B\} \rightarrow \{C\} \rightarrow \{D\} \rightarrow \{E\} \rightarrow \{F\}$, in that order. Often, it is hard for the guided simulation to transition through the required order of states in the concrete design to reach the target state, specially when the guide-points provide no indication as to what that order might be.

In the proposed approach, the behavior of the abstract model is constrained so that it more closely resembles the behavior of the concrete design, as illustrated in Figure 4.1. By doing so, we eliminate, the possibility of a large number of illegal transitions, such as $\{A, B, C, D\} \rightarrow \{F\}$, in the abstract model, (leaving only $\{E\} \rightarrow \{F\}$). The abstract preimages obtained in the constrained abstract model more closely resemble real preimages in the concrete design, and thus allow for much easier concretization than preimages obtained in the unconstrained abstract models.

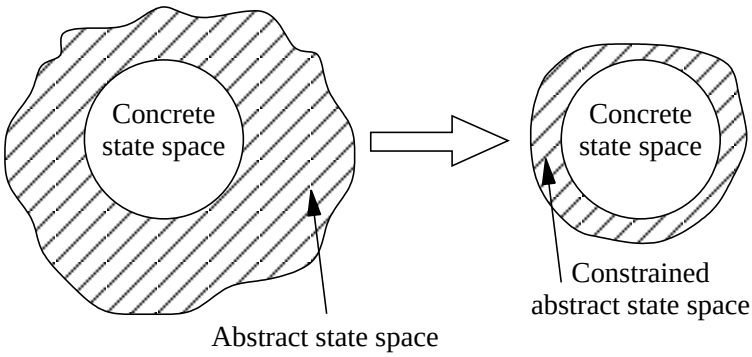

Figure 4.1: Abstraction Strengthening

To strengthen the abstract model, a set of unreachable partial states are quickly

identified in the concrete design. This set is further enlarged by performing resolution and cube enlargement. Then, the enlarged set of unreachable states is used to constrain the construction of preimages in the abstract model.

4.2 Star Algorithm

The *Star-Algorithm*[2] was originally proposed by Akers, Krishnamurthy, et al., for identifying inactive faults in combinational circuits for a given input vector. Given a combinational circuit, it generates a subset of circuit nodes whose values are sufficient for justifying the primary output (PO) values. The signals to a gate in the circuit that would be sufficient to produce the gate's output value are marked with a star (*). By iterating the marking process from the gates closest to the POs to primary inputs (PIs), we can identify which PIs could be relaxed without having changed the PO values. Thus, by only considering the PIs marked with a star, and considering all unmarked PIs as *don't care*, we get an *enlarged* PI cube, which is sufficient to justify the PO values. The following provides a brief description of the Star-Algorithm.

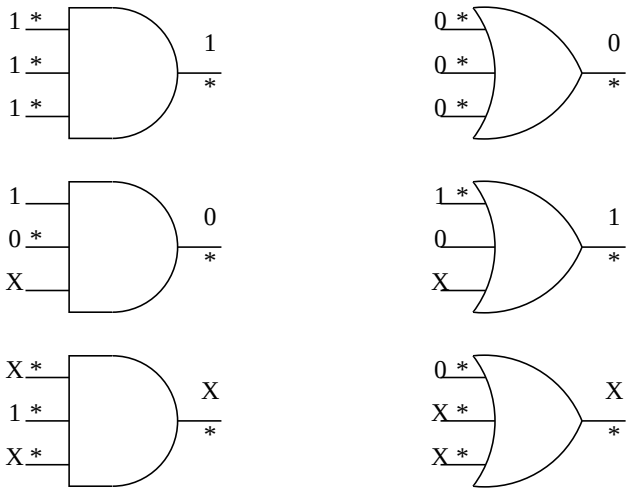

Figure 4.2: Placing of stars in the Star Algorithm

Stars (*) are initially placed on all PO nodes that are specified (i.e, not *don't care*), and the placing of stars is gradually moved backwards towards the PIs. Figure 4.2 shows the placement of stars on AND and OR gates. For AND gates with an output value 0, any 0-input is sufficient to fully specify the output value of the gate. A star is placed on that input. If there are more than one input with the value 0, the star may be placed at any of them arbitrarily. Similarly, for an OR gate with output value 1, a star is placed on one 1-input. On the other hand, if the output of the AND (OR) gate is 1 (0), then stars must be placed on all the inputs. Stars at the inputs of NAND and NOR gates are placed in similar manner, and stars are always placed on inputs of all other gate types. After the iterative placing of stars is completed for the entire circuit, those PIs not marked with a star can be set to *don't care*, since changing their value would not change the PO values for this particular input vector.

4.3 Identifying Unreachable Partial States

In this section, the method to quickly identify unreachable partial states in the concrete design is described. We want to identify *partial* unreachable states, with as few specified state variables as possible, so that the state cubes cover large expanses of the illegal state space of the concrete design. In addition, finding the sparsely specified cubes reduces the computation cost, since we effectively preclude calculation of redundant solutions which might be covered by the sparsely specified cube. These unreachable state cubes are then constraints in the abstract model to prevent it from wandering into the illegal state space of the design when computing abstract preimages of a target property. By doing so, spurious abstract traces are minimized, as the behavior of the abstract model is largely constrained to transitions that are truly feasible in the concrete design.

The overall method used to identify unreachable partial states in the concrete design is illustrated in Figure 4.3. In order to find an initial set of unreachable states, we first compute the support of all the state variables in the design. The *support*, S_i, of state variable s_i is

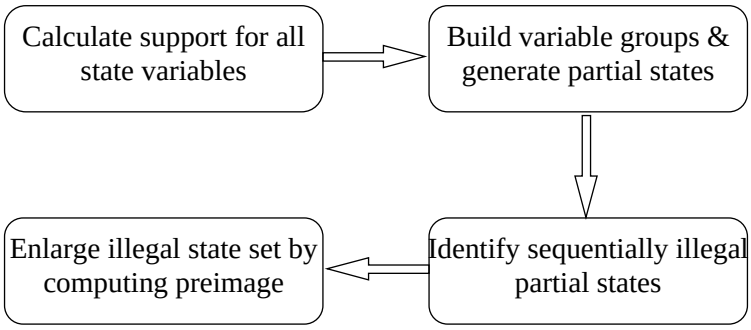

Figure 4.3: Unreachable State Identification Overview

the set of variables needed to determine the next state value of s_i in the transition relation. Once all the supports for every state variable in the concrete design have been computed, they are arranged in a matrix as showing in the Figure 4.4

$$\begin{bmatrix} a & & b & & & \\ b & a & & c & & f \\ c & a & b & & & \\ d & & & c & e & f \\ e & a & b & c & d & f \\ f & a & b & c & d & e \end{bmatrix}$$

Figure 4.4: Support matrix for each state variable

The rows of this matrix are sorted so that the variables with common support are grouped together in consecutive rows, as shown in Figure 4.5. This is done to ensure that those state variable that are more closely correlated are grouped together. Then, disjoint partitions of n consecutive rows are formed. The value of n is small, typically between 8 to 10 variables. For each partition, we enumerate all 2^n partial states for the variables in the partition, while all other state variables that do not belong to the partition are set to X (don't care). We collect and store all such partial states in a set P.

In order to determine if any of the partial states in P is unreachable, a simple bounded

$$\begin{bmatrix} f & a & b & c & d & e \\ e & a & b & c & d & f \\ d & c & e & f & & \\ b & a & c & f & & \\ c & a & b & & & \\ a & b & & & & \end{bmatrix}$$

Figure 4.5: Variables grouped according to common support

model checking (BMC) [3] instance is constructed to determine its unreachability via a SAT solver [22]. The concrete circuit is iteratively unrolled for k timeframes, and the output state variables are constrained to a specific target partial state $p \in P$. With the initial state variables left *unconstrained*, if the instance is unsatisfiable, it indicates that p is definitely an unreachable partial state, since no initial state can reach it. In such a case, we remove p from P and add it to state U_k. Note that as an unreachable partial state is computed, it can be used to constrain the initial state of the BMC instance to tighten the search for future unreachable partial states. Once all the properties in P have been tried, k is incremented and the procedure repeated until a maximum bound t is reached. As we have different values of k, various buckets of unreachable states, U_k, result at the end of this process.

Set U_k contains all partial states that have been determined to be sequentially unreachable in k steps. To enlarge this set of states, we perform resolution on the properties in set U_k and U_j where $1 \leq k, j \leq t$, and t is the maximum number of timeframes for which the BMC instance was unrolled. Resolving states that are unreachable in different values of k often yields new unreachable states that might be deeply embedded in the design [27]. Let R be the set of new states obtained after resolution was performed for all U_k, U_j. Thus, the complete set of all unreachable states identified so far, U, is given by

$$U = R \cup \bigcup_{i=1}^{t} U_i$$

Now the set U is further enlarged by computing it's one-step preimage. A 1-time-frame bounded model checking instance is setup and the output state variables are constrained to all the unreachable states in U. Since we want a new solution, one which is not already covered by any of the states in U, the initial state variables are also constrained to $\neg U$. A satisfying solution to this instance gives a new unreachable state, which is not covered by any of the states in U, in the initial state variables.

The solution cube thus obtained by the SAT solver is fully specified. We enlarge it using the Star Algorithm (described in section 4.2) as follows: A 1-time-frame unrolled iterative logic array (ILA) of the circuit is constructed, and the fully specified solution cube is assigned to the pseudo primary inputs (PPIs). The primary input (PI) vectors from the satisfying BMC instance extracted and logic simulated on the ILA. Since the output variables of the BMC instance were constrained to be within the set U, the fully specified state in the pseudo primary output (PPO) variables after logic simulation is covered by some state in U. The covering state is identified and assigned to the PPOs of the ILA. To enlarge the PPI cube, we perform the star algorithm for each specified PPO variable. Essentially, the star algorithm marks for each gate, those inputs that are *sufficient* to justify the gate's output value. All marked inputs receive a (*) marking. (Note that the PPO variables are not fully specified, because the covering state in U is a partial state). After the Star Algorithm has been performed for each specified PPO variable, the enlarged solution is obtained by considering only those PPI variables that are marked (*), and replacing all other unmarked PPI variables with don't care X. The enlarged solution cube is added to U and this process is repeated until the resulting instance is unsatisfiable, or a predetermined time limit is exceeded.

4.4 Computing the Strengthened Abstract Preimages

Given a set $A = \{A_1, A_2, ..., A_n\}$ of partitioned state variables, selected as described in Section 3.2.2, we compute the i^{th}-step *strengthened* abstract preimage for partition A_j in a

manner similar to that of the partitioned navigation tracks described in Section 3.2.3. We construct a BMC instance with the additional constraints of the unreachable states as follows: (i) The concrete design is unrolled for one time-frame and converted to a CNF formula. (ii) Constraints are added to the CNF formula as shown in the following expression:

$$(I \wedge \neg U_I \wedge \neg S_j * \wedge \hat{P}^i_{j-1}) \wedge (T_1 \wedge \neg U_1) \wedge (O \equiv S')$$

where I is the initial state variables, $\neg U_I$, $\neg U_1$ are the unreachable state constraints obtained from the preprocessing step described in Section 4.3, which are added to the initial state variables and the transition relation variables respectively, $\neg S_j*$ is the blocking-clause that prevents repeated computation of solutions already found for the current preimage, T_1 is the transition relation for the circuit, O is the output state variables, and S' is the set of target states. $\hat{P}^i_j$ tightens the preimage search by restricting the fully controllable initial state variables that appeared in abstraction sets $\{A_1, A_2, ..., A_{j-1}\}$ to values contained up until their respective i^{th}-step preimages, which were previously computed, and S' is the constraints on the output state variable. $\hat{P}^i_j$ and S' are defined in the same manner as in Section 3.2.3.

4.5 Experimental Results

The proposed unreachable state identification algorithm was implemented in C++. Experiments were conducted on ISCAS89 [17] and ITC99 [18] benchmark circuits, on a 3.2 GHz Pentium 4 with 1GB RAM, running Linux. The same logic simulation algorithm described in Section 3.3 was used in these experiments.

The unreachable state identification algorithm was implemented as a preprocessing tool, and allowed to run for 3600 seconds for each benchmark circuit. All illegal partial states discovered were recorded. Table 4.1 shows the number of illegal partial states identified, and the run-time taken by the tool for each benchmark.

Table 4.1: Identifying Unreachable Partial States

Circuit	# illegal partial states	Time (s)
s444	44	13.708
s526	60	10.829
s1423	31	157.67
s5378	32190	3600.00
b05	90	122.04
b11	637	157.18
b12	45414	3600.00
b13	2696	224.14

TO: 3600 seconds

We performed experiments to evaluate the effect of the illegal partial state constraints to strengthen the SAT instance when computing abstract preimages. For each benchmark, we picked a hard-to-reach state that was aborted by STRATEGATE [15]. We used this state as the target property. Twenty partitioned sets of state variables were generated using the GA based abstraction engine described in Section 3.2.2. Each partitioned set had a maximum of 10 state variables. The abstract preimage for each partition was computed to a fixed point, with and without the unreachable partial state constraints. Then, for each state s in the abstract preimage, we wanted to compute the shortest distance from s to the aborted state (target property). Figure 4.6 shows the average ratio of the distances to the aborted state between the strengthened instance and the un-strengthened instance. We observed a minimum of 17% lengthening of abstract traces for b12, while strengthened abstract traces for b05 were 70% longer. A longer abstract trace provides a finer distance metric to the simulator, as many abstract states which may have been clumped together in the same set in a shorter preimage, now appear in separate sets with different distances to the target via the longer trace.

We also compared the effectiveness of the abstraction-guided logic simulation with and without the preprocessing constraints added to the abstract model. The simulator was asked to find concrete traces to a number of particularly hard-to-reach properties for a maximum of 3600 seconds. Each abstraction consisted of eight state variables, and six such abstract

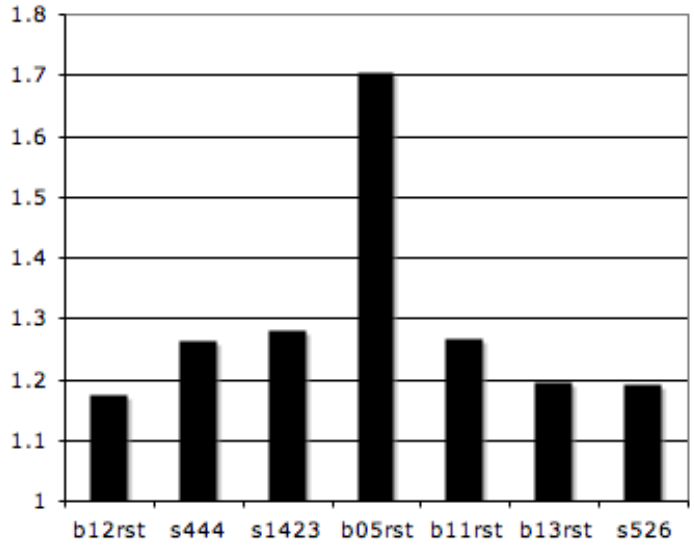

Figure 4.6: Ratio of strengthened to un-strengthened abstract distance

variable sets were selected for each target property using the GA algorithm described in Section 3.2.2.

Table 4.2 shows the results of our experiments. Each row represents a specific hard-to-verify aborted state. Column 1 lists the name of the benchmark and an aborted state, column 2 lists the runtime for pure random simulation, column 3 lists the time taken by the simulator to find a concrete trace to the target property using an abstract model without any constraints, and column 4 indicates the time taken when the abstract model was strengthened with the illegal partial state constraints.

Random simulation was not able to find a trace to any of the properties, indicating that none of the target properties was trivial. The results for the abstraction-guided simulators clearly illustrate the effectiveness of the constraints on the abstract model. For example, in s5378.3, abstraction guided simulation was able to reach the aborted state in 3472 seconds. When guided by the strengthened abstract model, the simulator is consistently able to concretize abstract traces faster than when the abstract model is left un-

constrained. For example,in both s5378.1 and s5378.2, without strengthening the abstract model, abstraction-guided simulation timed out at 3600 seconds; on the other hand, with our proposed abstraction-strengthening method, we were able to reach both of these two hard-to-reach aborted states. The preprocessing overhead of calculating the constraints was not added to the runtimes reported in Table 4.2, as they are a one-time cost. Once calculated, the constraints are utilized without additional cost, to all properties being verified. Further, we reckon the preprocessing cost is well worth the benefits, as in most cases it was found to be small. In s5378 and b12, the preprocessing phase took 3600 seconds. However, the simulator guided by the strengthened abstract mode was able to reach all target properties. On the other hand, when guided by an unconstrained model, the simulator timed out for most of the properties.

Table 4.2: Experimental Results

Property	Random	Abstraction-guided simulation	
		w/o constraints	w/ constraints
s382.1	TO	54.53	**32.89**
s444.1	TO	25.18	**23.79**
s526.1	TO	135.68	**112.94**
s5378.1	TO	TO	**2752.26**
s5378.2	TO	TO	**2194.17**
s5378.3	TO	3472	**2952.22**
b05.1	TO	75.21	**26.77**
b05.2	TO	124.72	**36.27**
b11.1	TO	114.29	**78.94**
b12.1	TO	TO	**1363.27**
b12.2	TO	2621	**2117**
b12.3	TO	TO	**3002.18**

TO: 3600 seconds

Chapter 5

Conclusion

Abstraction-guided simulation has proven to be a very promising verification framework, however current methods still find it difficult to find traces to hard-to-reach states. There are two key open-ended issues that have a great impact on the efficacy of this framework. (i) How does one construct the abstract model (ii) Overcoming artifacts of abstract, such as local optima and dead-end states.

In this thesis, we have addressed both these pressing issues, and presented a new and powerful abstraction-guided simulation framework to justify hard-to-reach target states in sequential circuits. To construct the abstract model, we employ a genetic algorithm to abstract state variables into partitioned sets, which are highly correlated to the target state. Since the abstraction is performed at the gate level, we require no high-level design information about the circuit. Our experimental results have shown that variables abstracted by our GA prove to be much more effective than randomly chosen variables. Next, the partitioned navigation track (PNT) is computed based on the abstracted partitioned sets. The PNT is used to guide the search to reach the target state. Since the PNT captures the abstract behavior of different portions of the state space, the simulator is provided with a much more refined view of the abstract state space. The fitness function of the GA based simulator effectively uses distance information from provided by each partition of the PNT.

This helps the simulator side-step local optima.

We also propose an abstraction strengthening preprocessing step, whereby preimages in the PNT are constrained, so as to make them much more closely resemble actual concrete traces. This further enables the logic simulator to obtain very accurate distance metrics to guide it's search towards the target. Our method employs a preprocessing step to quickly identify a set of unreachable partial states in the concrete design. This set of states is enlarged by performing resolution, cube enlargement, and preimage computation. The resulting set of unreachable partial states prove to be powerful in constraining the behavior of an abstract model of the design. Abstract traces obtained from the strengthened model are longer than those obtained in from an unconstrained model, because the constraints prevent the model from wandering into the illegal state space of the design. Thus the simulator is able to much more effectively distinguish between concrete states encountered during simulation, as two concrete states which might have had the same abstract distance to the target in the unconstrained model, are likely to map to abstract states with different distance values in the lengthened trace.

Lastly, our method requires no refinement to the abstraction. The cost of abstraction refinement is exponential in the number of state variables refined. Thus the time and space costs required in our approach is greatly reduced, allowing us to tackle large designs effectively. Experimental results showed that our technique is very effective in reaching hard-to-reach states where previous methods fail.

Bibliography

[1] V.D. Agrawal and S.T. Chakradhar. Combinational atpg theorems for identifying untestable faults in sequential circuits. *Proc. European Test Conf.*, pages 249–253, 1993.

[2] Sheldon B. Akers, Balakrishnan Krishnamurthy, Sungju Park, and Ashok Swaminathan. Why is less information from logic simulation more useful in fault simulation? In *ITC '93: Proceedings of the International Test Conference*, pages 786–800, 1993.

[3] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[4] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 46–51, New York, NY, USA, 1990. ACM Press.

[6] Pankaj Chauhan, Edmund M. Clarke, James H. Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, pages 33–51, London, UK, 2002. Springer-Verlag.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[8] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[9] Edmund M. Clarke, Anubhav Gupta, James H. Kukula, and Ofer Shrichman. Sat based abstraction-refinement using ilp and machine learning techniques. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 265–279, London, UK, 2002. Springer-Verlag.

[10] Malay K. Ganai, Adnan Aziz, and Andreas Kuehlmann. Enhancing simulation with bdds and atpg. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 385–390, New York, NY, USA, 1999. ACM Press.

[11] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[12] Saurav Gorai, Saptarshi Biswas, Lovleen Bhatia, Praveen Tiwari, and Raj S. Mitra. Directed-simulation assisted formal verification of serial protocol and bridge. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 731–736, New York, NY, USA, 2006. ACM Press.

[13] Pei Hsin Ho, Thomas Shiple, Kevin Harer, James Kukula, Robert Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. Smart simulation using collaborative formal and simulation engines. In *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 120–126, Piscataway, NJ, USA, 2000. IEEE Press.

[14] J. H. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

[15] Michael S. Hsiao, Elizabeth M. Rudnick, and Janak H. Patel. Sequential circuit test generation using dynamic state traversal. In *EDTC '97: Proceedings of the 1997 European conference on Design and Test*, page 22, Washington, DC, USA, 1997. IEEE Computer Society.

[16] Alan J. Hu. Formal hardware verification with BDDs: An introduction. In *Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*, pages 677–682. IEEE, 1997.

[17] Iscas'89 benchmark circuits. `http://www.cad.polito.it/tools/tar/iscas-gate.tar.gz`.

[18] Itc'99 benchmark circuits. `http://www.cad.polito.it/tools/itc99.html`.

[19] Andreas Kuehlmann, Kenneth L. McMillan, and Robert K. Brayton. Probabilistic state space search. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 574–579, Piscataway, NJ, USA, 1999. IEEE Press.

[20] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[21] Hari Mony, Jason Baumgartner, Viresh Paruthi, Robert Kanzelman, and Andreas Kuehlmann. Scalable automated verification via expert-system guided transformations.

[22] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[23] Kuntal Nanshi and Fabio Somenzi. Guiding simulation with increasingly refined abstract traces. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 737–742, New York, NY, USA, 2006. ACM Press.

[24] Irith Pomeranz and Sudhakar M. Reddy. On identifying undetectable and redundant faults in synchronous sequential circuits. *Proc. VLSI Test Symp.*, pages 8–14, 1994.

[25] Smitha Shyam and Valeria Bertacco. Distance-guided hybrid verification with guido. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1211–1216, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[26] J. P. Marques Silva and Karem A. Sakallah. Grasp–a new search algorithm for satisfiability. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[27] Vishnu C. Vimjam and Michael S. Hsiao. Fast illegal state identification for improving sat-based induction. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 241–246, New York, NY, USA, 2006. ACM Press.

[28] Dong Wang, Pei-Hsin Jiang, James Kukula, Yunshan Zhu, Tony Ma, and Robert Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 35–40, New York, NY, USA, 2001. ACM Press.

[29] Qingwei Wu and Michael S. Hsiao. State variable extraction to reduce problem complexity for atpg and design validation. In *ITC '04: Proceedings of the International Test Conference on International Test Conference*, pages 820–829, Washington, DC, USA, 2004. IEEE Computer Society.

[30] Praveen Yalagandula, Vigyan Singhal, and Adnan Aziz. Automatic lighthouse generation for directed state space search. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 237–242, New York, NY, USA, 2000. ACM Press.

[31] C. Yang and D. Dill. Spotlight: Best-first search of fsm state space, 1996.

[32] C. Han Yang and David L. Dill. Validation with guided search of the state space. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 599–604, New York, NY, USA, 1998. ACM Press.

[33] Jun Yuan, Jian Shen, Jacob A. Abraham, and Adnan Aziz. On combining formal and informal verification. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 376–387, London, UK, 1997. Springer-Verlag.

Vita

Ankur Parikh was born in Mumbai, India. After completing secondary education at the Cathedral and John Connon School in Mumbai, he obtained a Bachelor of Science degree in Computer Engineering from New Jersey Institute of Technology in 2004, following which he worked as a firmware engineer at Datascope Corp., NJ. He joined the Master of Science program in the Electrical and Computer Engineering department at Virginia Tech in the fall of 2005. He joined Dr. Michael Hsiao's PROACTIVE research group in May 2006, and has since been involved in research related to semi-formal verification.

Permanent Address: 34 Minoo Desai Road
Mumbai 400005
India