Chapter 1
Introduction

Validation dominates the design cycle time for large digital integrated circuits [77]. The volume of time and resources that are typically dedicated towards design validation has grown with the increasing size and complexity of designs [96], and yet it has become increasingly hard to arrive at a satisfactory verification closure. Of particularly annoying nature are the bugs that are detected late in the design cycle, since they are hard to fix and even harder to verify that the fix actually covers all related bug scenarios [62]. This thesis aims to study some of these verification problems.

Large digital integrated circuits typically consist of several functional blocks (components) and a non-trivial amount of *glue logic* which stitches the components together through well defined control and data paths. The glue logic implements the architectural specification of the design using (and often reusing) the core functional units of the design. It is therefore not surprising that a non-trivial fraction of the late bugs are found in the glue logic, and most of these bugs arise out of incorrect interpretation or implementation of the micro-architectural specifications [40]. When such bugs are found late in the design flow, the designer has to fix the bug and then verify that the bug fix is correct and complete.

Quite often, the bug fixes are local in nature, that is, the part of the design which is modified, is of modest size and is contained within a small structural boundary [62]. However, the cone of influence of this patch can be significantly larger and in many cases may touch the entire architecture. The size (and complexity) of this cone makes the problem of verifying the correctness of the fix one of the most challenging tasks in design verification. The focus of this thesis is on verification problems of this nature.

Traditional approaches for verifying the correctness of late bug fixes in the glue logic rely on simulation coverage of the scenario that exposed the bug and possibly related scenarios. Unfortunately, at this level of integration it is very hard to determine and cover all possible scenarios that may expose the same bug (or a similar one) through simulation. Consequently it is not entirely uncommon

in industrial practice that the bug fix does not cover all the scenarios and a very similar bug is uncovered again in the future.

With the increased adoption of *assertions* (formal properties) by the digital chip design industry, it has become a part of recommended practice to add an assertion to capture the intended behavior when a bug that misrepresents the intent is found. Our challenge can therefore be articulated in terms of two verification problems, namely (a) to verify that the design, after the (local) changes, satisfies the new assertion under all possible scenarios, and (b) to verify that the change does not affect the functionality of those portions of the design where no change is intended. This thesis proposes to use several novel techniques in conjunction with formal property verification for addressing the first verification problem from a practical industrial perspective, and novel methods for sequential equivalence checking for addressing the second verification problem.

1.1. Motivation and Objectives

The problem addressed in this thesis can be articulated with the help of Figure 1.1. $\mathcal{B}$ represents the fragment of logic that has been modified to fix the bug. $\mathcal{I}_\mathcal{B}$ and $\mathcal{O}_\mathcal{B}$ represent the inputs/outputs into/from $\mathcal{B}$ respectively. The property, $\mathcal{P}$, represents the intended local behavior, which has been added after the bug was found. Our goal is to prove that $\mathcal{B}$ satisfies $\mathcal{P}$ under all possible scenarios that may be created by the glue logic, $\mathcal{D}$. The main challenge in this problem is to determine which scenarios can be created by $\mathcal{D}$, and verify whether $\mathcal{B}$ satisfies $\mathcal{P}$ in each of them.

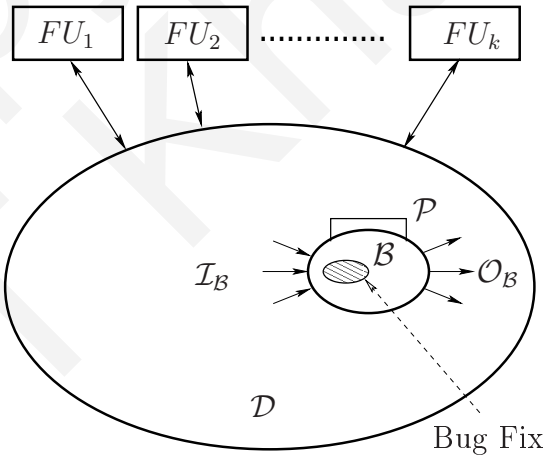

Figure 1.1: Problem scenario

It is typically infeasible in industrial practice to verify $\mathcal{P}$ on $\mathcal{D}$ using formal verification techniques [70] due to the enormous size of the glue logic, $\mathcal{D}$. On the other hand, formal methods can be used to verify $\mathcal{P}$ on $\mathcal{B}$ (which is much smaller). If $\mathcal{B}$ satisfies $\mathcal{P}$, then obviously $\mathcal{D}$ satisfies $\mathcal{P}$ as well, but the reverse

is not necessarily true. It is possible that none of the scenarios under which $\mathcal{B}$ refutes $\mathcal{P}$ can be created by $\mathcal{D}$. In other words, factoring in the influence of the entire glue logic on the verification of $\mathcal{P}$ on $\mathcal{B}$ is our main challenge.

There is a significant volume of literature on incremental verification of digital designs. In [78, 142, 143, 147, 148], the problem of incremental verification of hardware designs is viewed as an instance of dynamic graph algorithms. In these works, the design is represented as a graph and changes to the design are viewed as edge insertions or deletions. After each such graph update, the problem is to check whether properties which previously held on this graph, still hold and therefore this can be viewed as a problem of reachability analysis on dynamic graphs. However dynamic graph connectivity is an open problem with known scalability limitations [67]. An incremental and complete bounded model checking algorithm has been proposed in [95] using incremental satisfiability solving. But our experience shows that bounded model checking does not scale to the designs we focus on. Recently, Bradley et al. have proposed the ic3 model checker, which is an incremental, inductive model checker [46, 49], using the incremental generation of stepwise-relative inductive clauses. It seems to be a promising new direction in symbolic model checking and is amenable to incremental analysis. In [67], Chockler et al. have used the same ic3 model checking algorithm to propose an incremental verification algorithm that stores information from previous verification runs and reuses it, in case of both positive and negative verification results. However these methods are applicable when a property is formally proven on the original design, whereas in our problem the property $\mathcal{P}$ is added after the bug is found and model checking $\mathcal{P}$ on $\mathcal{D}$ does not scale.

Combining formal and simulation techniques to achieve a more complete verification of a system [43] has attracted a lot of attention. In [98], a tool Ketchum has been proposed which combines simulation and formal-based techniques to achieve better verification through automatic test generation and reachability analysis. In [157], algorithms which combine simulation with symbolic methods for the verification of invariants are proposed. In [94], methodologies have been presented to relate simulation and formal property verification techniques more formally to achieve a potentially better strategy for cohesive coverage management in verification.

Intuitively it may appear that counterexample guided abstraction refinement approaches [75, 110] are useful for our problem. These methods start with a small cone of state variables and progressively expand the cone by including state variables needed to eliminate false counterexamples. The task of determining whether a counterexample is real/false typically works well when the property is defined on the boundary of the entire logic, because the counterexample defines the

inputs to be driven to test whether the counterexample is reproduced in simulation. On the other hand, in our problem, the counterexample is defined on the signals of $\mathcal{B}$, and there is no obvious way of determining whether it is real in the entire glue logic $\mathcal{D}$.

The motivation of this thesis is to leverage formal methods to aid the existing manual approaches for verifying bug fixes. Instead of attempting to use formal methods as a singular technology for the verification problem described above, this thesis aims to use formal methods to improve coverage of relevant behaviors and restrict the attention of the verification engineer to those behaviors that are relevant for the property. Specifically this thesis addresses the following broad objectives:

1. Traditionally, after a bug fix, the verification engineer simulates the design with the same test-bench that was used to expose the bug. Formally, this test-bench executes one control path of the design under a specific assignment to the data variables. The bug may resurface on the same control path for a different valuation of the data variables, or it may resurface on a different control path altogether. The objective of this thesis is to classify these possibilities formally and then leverage the given test-bench to determine which of these are possible. If the bug fix is not *robust*, that is, it cannot be proven formally using local information, then the objective is to make the verification engineer aware of the gaps in the verification.

2. Attempting to prove $\mathcal{P}$ on $\mathcal{B}$ is always a recommended step since its success achieves verification closure for $\mathcal{P}$. On the other hand, when $\mathcal{B}$ refutes $\mathcal{P}$, then the model checker (that is, the formal verification tool) may return fictitious counterexamples. If a real bug persists, then the real counterexamples may not be easily distinguishable from the numerous false counterexamples. It is not feasible in industrial practice for the verification engineer to examine all counterexamples and determine their correctness. The objective of this thesis is to extract knowledge from the large volume of existing simulation data (including those for individual components) to filter out the counterexamples that have very little support from simulation data, and then use the knowledge extracted to rank the counterexamples in terms of their likelihood of being real.

3. A bug fix can often lead to side-effects, that is, it can inadvertently affect parts of the design where no change was intended. It is therefore necessary to prove that those signals that are supposed to remain the same indeed preserve their functionality in the modified design. Since the bug fix is local

in nature, it does not affect the local invariants of the other components within the glue logic. One objective of this thesis is to leverage the local invariants of these other components to prove sequential equivalence between a signal in the original design and the same signal in the modified design.

For each of these problems, our goal is to provide the verification engineer with analytical aids for increasing his/her confidence on the verification of the bug fix. We believe that this is a genuine requirement in practice considering the absence of any stand-alone formal method for this problem.

1.2. Summary of Contributions

This thesis presents the findings of our research on developing methodologies for aiding verification of local design changes in industrial-size digital integrated circuits and demonstration of the effectiveness of our methodologies on various test cases. This section presents a summary of the major contributions.

1.2.1. Trace Assisted Formal Methods for the Verification of Bug Fixes

When a bug is uncovered in the design late in the design cycle, the design engineers fix the bug and simulate the modified design with the same test-bench that exposed the bug, to determine whether the given bug is now eliminated. But this does not guarantee that the bug cannot resurface through similar but slightly different scenarios. In this part of the thesis, we define a family of related bugs, given a bug manifestation, by identifying a minimal dynamic causal slice that affects the property $\mathcal{P}$. We analyze the logic flow inferred by this dynamic causal slice with respect to the conditions required to satisfy $\mathcal{P}$. Using this analysis, given a fix for the bug, we provide a categorization of the bug fix, whereby we guarantee the complete elimination of the bug in case it is a robust fix and in other cases, provide a report of other counterexamples or the gaps in coverage of this verification run, as appropriate. Specifically the problem we address is as follows:

Definition 1.1. [Problem Statement]:
Given:

1. *$\mathcal{D}$: The RTL implementation of the original glue logic.*
2. *$\mathcal{B}$: A module contained within $\mathcal{D}$.*
3. *$\mathcal{P}$: A property over the signals at the boundary of $\mathcal{B}$.*
4. *$\mathcal{B}'$: The modified version of module $\mathcal{B}$.*

5. $\mathcal{D}'$: *The version of* $\mathcal{D}$ *containing* $\mathcal{B}'$ *instead of* $\mathcal{B}$.

6. $\mathcal{TB}$: *The test-bench driving* $\mathcal{D}$ *which demonstrated the bug.*

Goal:

- *To determine whether the modification of* $\mathcal{B}$ *to* $\mathcal{B}'$ *corrected the bug scenario.*

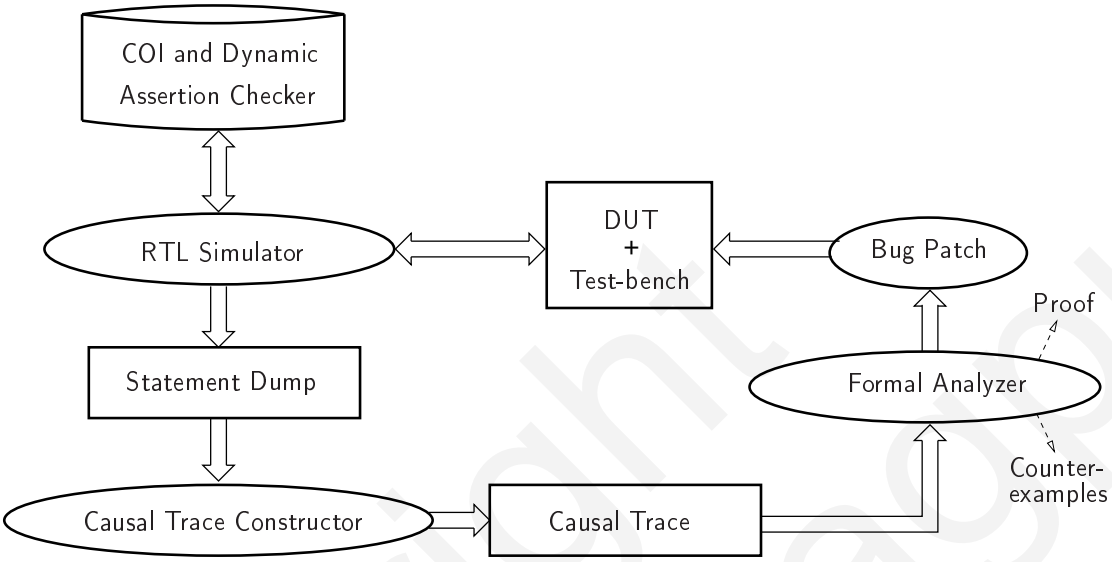

Figure 1.2: Overall methodology Flow

The original design had $\mathcal{B}$ inside $\mathcal{D}$. Since $\mathcal{TB}$ was used to demonstrate the bug, and $\mathcal{P}$ was added in hindsight, it follows that the simulation trace obtained by driving $\mathcal{D}$ with $\mathcal{TB}$ refutes $\mathcal{P}$. When $\mathcal{B}$ is replaced with $\mathcal{B}'$ to patch the bug, it is normal practice to simulate $\mathcal{D}'$ with $\mathcal{TB}$ again and check that the trace satisfies $\mathcal{P}$. We propose to extend this step with some additional formal methods inspired by software verification and debugging to analyze the new trace and check whether the proof of $\mathcal{P}$ on similar scenarios follows from it.

The broad steps in the proposed approach are as follows (see Figure 1.2):

1. We simulate $\mathcal{D}'$, which is the modified glue logic containing the bug fix, using $\mathcal{TB}$ and dump the set of statements $\mathcal{S}$, which *causally affect* a *non-vacuous* satisfaction of $\mathcal{P}$.

2. We construct a dynamic causal slice $\mathcal{C}$ from $\mathcal{S}$ to isolate all statements which affect the success of $\mathcal{P}$ through data or conditional control dependencies. We call this slice the *causal trace*.

3. While traversing the execution trace during dynamic slicing, we also compute the *unrestricted weakest precondition* (WP) $\mathcal{Q}$ along the slice. The weakest pre-condition is only computed for the statements which are in the dynamic

slice. It terminates when the slicing terminates. The WP thus computed is the logic inferred by the execution flow obtained in simulation.

4. Based on the WP, we categorize the bug fix into one of the following three types and accordingly provide the following information:

   (a) *Type-1 Fix*: This is the kind of fix for which the bug cannot resurface either on another control flow or with different data inputs on the same control flow. This can be intuitively explained as follows. In the same scenario, in any run, the same set of statements will be executed. These statements may have some branch conditions whose values are determined from previous statements that are not causally determined from the values dumped during this window. If the fix ensures that there are no such branch conditions, then we have truly covered the scenario and the fix is robust. We can thus provide a guarantee in this case that the bug is completely eliminated in this scenario.

   (b) *Type-2 Fix*: This is the kind of fix for which the bug cannot resurface with differing data input valuations on the same control path as simulated by the given test-bench, but might resurface through some alternative control flow of the RTL code. We guarantee that the bug cannot resurface with differing data input valuations on the same control path as simulated by the given test-bench and additionally report the control points (conditional expressions) in the slice, which are partially covered and exactly which part of each control point is covered by this verification run.

   (c) *Type-3 Fix*: This is the kind of bug fix where the bug can resurface on the same control flow for differing valuation of data inputs. Given such a fix, we report another counterexample of $\mathcal{P}$.

We provide experimental evidence to demonstrate that the complexity of verifying a bug fix with respect to the given bug scenario is orders of magnitude less than attempting to prove $\mathcal{P}$ on $\mathcal{D}'$. We demonstrate this on the snoopy Cache Coherence protocol of Pentium Pro (P6), an implementation of which can be found in the "p6bus" example in the Texas 97 benchmark suite [35]. We also demonstrate the effectiveness of our methodology on the L2 cache control logic of the industrial-size open-source OpenSPARC T1 processor [18]. Bounded model checking of $\mathcal{P}$ does not scale on the L2 cache of OpenSPARC due to capacity limitations. But given a bug manifestation and a bug fix for the same logic, we are able to perform the aforementioned formal analysis in around 5 hours of time, and with reasonable amount of memory.

1.2.2. Formal Methods for Ranking Counterexamples

For bug fixes that are not *robust* (as outlined in Section 1.2.1), we must determine whether $\mathcal{P}$ holds on $\mathcal{D}$, as shown in Figure 1.1. Since $\mathcal{B}$ is small, it is feasible to formally verify $\mathcal{P}$ on $\mathcal{B}$ in isolation, that is, under the assumption that the inputs to $\mathcal{B}$ are unconstrained. If $\mathcal{P}$ holds on $\mathcal{B}$ in isolation, then $\mathcal{P}$ also holds on $\mathcal{D}$, however the reverse is not true. For example, a counterexample generated by model checking $\mathcal{P}$ on $\mathcal{B}$ in isolation may not exist in $\mathcal{D}$, that is, $\mathcal{D}$ may not be able to create the scenario depicted by the counterexample on the interface of $\mathcal{B}$. Typically the spurious counterexamples are numerous and often exceed the number of real counterexamples, if any. It is practically infeasible for the designer to manually examine each counterexample and determine whether it is real or spurious within the large glue logic $\mathcal{D}$. Formally verifying whether a counterexample over $\mathcal{B}$ is real in $\mathcal{D}$ is infeasible in practice due to the size of $\mathcal{D}$ and the fact that the counterexample is defined over signals which are not on the interface of $\mathcal{D}$.

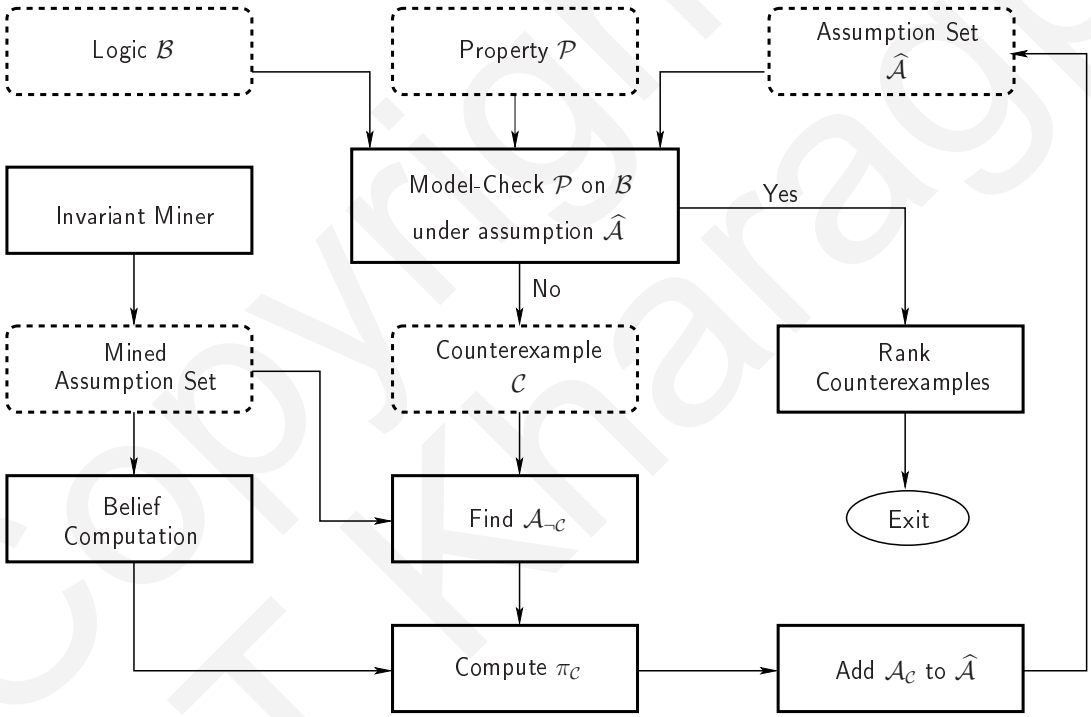

Figure 1.3: Counterexample ranking tool flow

In this part of the research, we present a method for ranking the counterexamples obtained from model checking $\mathcal{P}$ on $\mathcal{B}$, using knowledge mined from the large volume of existing simulation dumps on $\mathcal{D}$. The expectation of the proposed approach is that higher ranked counterexamples are more likely to be real than the lower ranked ones. The proposed approach for ranking counterexamples consists of the following steps (see Figure 1.3):

1. *Assumption mining.* We use assertion mining tools [114, 154] on simulation

traces of $\mathcal{D}$ to create a large set of possible *assume properties* over the interface of $\mathcal{B}$. The assumptions are mined with the goal of capturing the effect of $\mathcal{D}$ on the interface of $\mathcal{B}$. The assertion miner is biased accordingly.

2. *Assumption weighting.* We assign a *belief* value to each mined assume property based on the evidences received in support of the assume property from the simulation traces. These weights are real values between zero and one, and thereafter treated as confidence measures for the assume properties.

3. *Counterexample ranking.* We group counterexamples based on the sets of assume properties they contradict ($\mathcal{A}_{\neg\mathcal{C}}$). We aggregate the belief values for the assume properties in the conflict set to compute a confidence metric $\pi(\mathcal{C})$ for the counterexample. This confidence metric is used to rank the counterexamples. Intuitively, counterexamples which contradict assume properties of high support are ranked lower than the others. Counterexamples which do not have any conflict with the assume properties are ranked the highest. Finally counterexamples are presented to the verification engineer in descending order of rank.

The key requirement in the third step is to be able to rank the counterexamples *without actually generating all of them through model checking.* This is achieved by grouping the counterexamples based on the sets of assume properties they contradict and *by ensuring that a counterexample from the same family is not generated again while searching for counterexamples having higher confidence.* Here $\mathcal{A}_{\mathcal{C}}$ represents the disjunction of the assume properties in $\mathcal{A}_{\neg\mathcal{C}}$, which is added back to assumption set $\widehat{\mathcal{A}}$.

In the second part of this research, we look at the same problem of assigning confidence values to counterexamples in the case that the available simulation traces are not global traces on the whole of $\mathcal{D}$, but scattered simulation traces on individual modules surrounding $\mathcal{B}$. In this case, we cannot directly mine assume properties on the interface of $\mathcal{B}$ from the scattered simulation traces. We need to determine the dependences between the signals present on the interfaces of the different surrounding modules of $\mathcal{B}$ and also the dependences between the different mined assumptions from different modules. We proceed to assign a confidence to any given counterexample trace, using probabilistic reasoning as follows:

1. We mine assume properties on the output signals of each surrounding module of $\mathcal{B}$, in terms of its interface signals.

2. We construct a Bayesian network [82], in which nodes represent $\langle var, time \rangle$ tuples, where *var* is a signal present in a mined assumption and *time* is an integer that varies from 0 to the length of the counterexample trace.

3. We add the edges of the Bayesian network according to the temporal dependences between the signals which are present in the mined scattered assumptions, considering each of them, one at a time. This completes the structure of the Bayesian network.

4. The conditional probability tables are constructed by evidences received in support of the particular value combinations of the signals from the respective simulation traces. These weights are normalized to real values between zero and one. We add the prior probabilities of the valuations of these signals also from the traces.

5. Now we query the Bayesian network for the posterior probability that it assigns to the valuations of signals present in the given counterexample trace. We assign that probability to be the confidence of this counterexample.

We demonstrate both parts of our counterexample ranking methodology on large-sized academic benchmark circuits taken from the ISCAS 89 benchmark suite [13] and also on the L2 cache control logic of the industrial-size open-source OpenSPARC T1 processor [18]. The experimental results show a good correlation between the confidences assigned to the counterexamples and the actual truth of the counterexamples.

1.2.3. Reusing Component Invariants in Equivalence Checking

When the glue logic is modified for a bug fix, the designer hopes that the side effects of the fix are locally contained, that is, the bug patches made in the glue logic will not affect equivalence between signals which were not intended to be modified. However the designer is never sure which signals are logically affected by the patch in the context of the entire glue logic. Typically bug fixes can often involve micro-architectural change to the design, involving timing changes, changes in pipelining, pre-computing of values or such other sequential modifications due to which the mapping between the flops in the original version of the design and those in the modified version cannot be established. Therefore, after every patch, a formal sequential equivalence checking has to be performed between a signal in the modified logic and the corresponding signal in the original logic, if that signal is not intended to be affected by the patch. This equivalence checking cannot be restricted to the boundaries of the components which were patched and must typically consider the entire glue logic. But sequential equivalence checking (SEC) [79, 80] involves the traversal of the state space of the product machine of the original and modified versions and as such, optimizations need to be applied to it, in order to make it scalable to industrial-size designs.

It is obvious that the components that were not touched by a patch remain logically equivalent in isolation. *Can we use this fact to prune the search during equivalence checking?* Since the modified parts of the glue logic can constrain the inputs to an untouched component in a different way as compared to the original circuit, it is possible that the logic behind a signal in an untouched component is no longer equivalent to the original circuit in the context of the entire glue logic. Therefore, the search for equivalence checking of the glue logic cannot completely prune the logic of untouched components. However, we show in this part of our research, that we can significantly prune the state space by using the equivalence between the untouched components in the original and modified circuits.

To check whether the functionality of a signal in the modified design is equivalent to that in the original design, these two signals are tied through XOR gates, and the corresponding inputs to the design are tied together. Thereafter, SEC methods are used to check whether the output of the XOR gate can ever be 1. This basically boils down to checking whether there is any state in the reachable state space of the composite design, which turns the XOR gate on. Therefore a global invariant in our SEC context is the property that the output of this XOR gate should always be 0, or in other words, should never be 1.

Target enlargement techniques [38], solve this problem of SEC by starting with the target property φ to be the property that the output of an XOR gate is 1. Target enlargement does backward state space traversal from the set of states satisfying φ, attempting to prove that the initial state is not reachable. If the initial state is reached, then a state satisfying this target (namely, output of XOR gate is 1) is reachable from the initial state, and consequently the two signals are not equivalent. If the initial state is not reached, but a fixpoint on the size of the target is reached, then we can conclude that the two signals are equivalent. In this part of the thesis, we focus on sequential equivalence checking through backward state space traversal of the product machine, specifically target enlargement.

In a component-based design, once a component is proved to be equivalent to its model, we have a set of known unreachable states of that component state space, from that individual component's fixpoint. The state spaces are represented implicitly using Binary Decision Diagrams (BDDs) and the BDD representing the set of unreachable states due to all unchanged components of the glue logic, constitutes a don't care function BDD, because the component unreachable states also remain unreachable in the global state space and transitions from the unreachable states can be manipulated to simplify the global transition relation of the design. Specifically, in this part of the thesis we do the following:

1. Our methodology takes advantage of these known unreachable states of the pre-verified components, to simplify the SEC of the modified design against

the original design, which typically involves a prohibitively large state space, being the product of the state spaces of the components. So the local invariants that are being utilized here basically represent the pre-verified information that the unpatched components are equivalent to their original versions.

2. Using the BDD of the unreachable states of components as don't care, we simplify the transition relation BDD of the entire glue logic, *while* it is being constructed, so that at no point of time the un-optimized global transition relation BDD has to be accommodated in memory, which creates a memory bottleneck due to its prohibitive size.

3. When the original design and the modified design remain equivalent even after the local change, then we are able to reduce the sequential depth of the search by dropping any unreachable state that is encountered at any step of the backward search.

Experimental results on academic benchmarks and also larger industrial benchmarks and comparison with an existing academic sequential equivalence checking solution demonstrate the effectiveness of our methodology in terms of the global transition relation BDD size and the amount of time taken for proving equivalence. We have obtained an average reduction of 39% in transition relation BDD size using our methodology on the ISCAS 89 [13] and ITC 99 [14] benchmark circuits and an average reduction of 37% in transition relation BDD size on the HWMCC 08 [7] benchmark circuits and a significant reduction in sequential depth and time taken. Our methodology is also established to perform better in terms of running time than ABC [2] for proving equivalence of the larger circuits when the component invariants are reused.

1.3. Organization of the Thesis

The thesis is organized as follows.

Chapter 1: This is the introductory chapter which discusses the objective and motivation of the problems addressed in the dissertation.

Chapter 2: This chapter presents background literature in the area of this research.

Chapter 3: This chapter presents a methodology for trace assisted verification of hardware bug fixes which involve a localized design change.

Chapter 4: In this chapter, we introduce the methodology of ranking counterexamples thrown up by local model checking on a component of the glue logic, making use of the assumptions mined from simulation traces on the glue logic.

Chapter 5: In this chapter, we present a case study on the OpenSPARC T1 processor L2 cache and demonstrate the methodologies discussed in Chapters 3 and 4 on this.

Chapter 6: In this chapter, we address the problem of sequential equivalence checking of the glue logic of a DUT after having undergone a patch, with a model which is typically a previous version of the same glue logic without the patch, by reuse of proved component invariants.

Chapter 7: This chapter summarizes the contribution of this research and presents outlines of future problems.