HARMONIZING DATA MINING AND STATIC ANALYSIS TO TACKLE HARDWARE AND SYSTEM LEVEL VERIFICATION

BY

LINGYI LIU

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:
Assistant Professor Shobha Vasudevan, Chair
Professor Jiawei Han
Professor Wen-mei Hwu
Professor Rob Rutenbar
Verification continues to pose one of the greatest challenges for today’s chip design. Formal verification and simulation-based verification have been widely adopted. Both of them always rely on assertions (a.k.a. properties) to express a design’s intended behaviors. State-of-the-art formal verification suffers from the scalability issue, and the simulation based method does not suffice in covering design behaviors. Moreover, the used assertions are always manually written, which greatly lowers the usability of hardware verification. For complex system on chip (SoC) design, the electronic system level (ESL) methodology creates a high level abstract view of the design, and the model is used for verifying functionality and performance at the early stage. Due to the fast simulation speed and high model complexity, the entire verification methodology is unsystematic and ad hoc, and lacks the support from EDA tools.

In this dissertation, we first present two systematic input stimulus generation methods for simulation based verification of register transfer level (RTL) design. The first method is based on STAR, a technique for generating input vector patterns for all paths of an RTL design using RTL symbolic execution. To attack the path explosion problem in STAR, we present HYBRO and the symbolic state caching method. HYBRO uses branch coverage metric to guide the path exploration. It is a best-effort method that produces excellent coverage for practical designs. Symbolic state caching considers the reachable state space when exploring different paths. It only generates tests for paths leading to previously uncovered state space. As a result, the path explosion problem is mitigated, and the entire method is much more scalable than the original STAR.

Another method for simulation based verification is based on GoldMine, an automatic assertion generation tool that was developed at the University of Illinois. We use the counterexamples generated from formal verification of GoldMine assertions to incrementally refine the internal decision tree. The counterexamples in every iteration can deterministically increase the coverage of the design and eventually cover
the whole reachable input space.

To improve the quality of generated assertions, we present a novel technique that combines static and dynamic analysis of RTL source code to discover word level features for assertion mining algorithm. That allows the mined assertions to be at the same level of abstraction as RTL instead of the Boolean bit level. The machine learning algorithm, as such, is thus agnostic to the level of abstraction of its features.

For ESL verification, we present our technique for generating assertions from transaction level models (TLMs) using GoldMine. The generated assertions, which are in the form of frequent patterns in simulation traces of TLMs, are able to express functionality specification as well as performance specification. Our static analysis technology also guides the mining algorithm to generate assertions capturing the data propagation relationships among function parameters and return values in TLMs. We attempt two mining algorithms for TLM assertion generation: sequential pattern mining and episode mining. We demonstrate that episode mining is more scalable and generates higher quality assertions than does sequential pattern mining.

Diagnosing performance violations in ESL verification is still a manual and time-consuming process in industry. We present an intelligent method to localize root causes of performance violations from simulation traces. We propose a concurrent mining method to discover concurrent patterns from the traces, which are potential root causes of the violations. We apply three categories of domain knowledge to increase the effectiveness of the mining results. We show that the concurrent pattern mining with domain knowledge pinpoints the root cause of a violation to a few patterns among transaction traces of massive size.
To my family
For enduring my absence in the past years
ACKNOWLEDGMENTS

As I have reached the stage of writing my final dissertation, I need to look back over the entire journey of my Ph.D. study. Although only my name appears on the cover of this dissertation, I have been fortunate to gain the advice, encouragement, and support from a lot of people in the last four years. I would like to take advantage of this space to gratefully acknowledge their help.

First and foremost, I would like to express my profound gratitude to my advisor, Prof. Shobha Vasudevan. I have always felt very lucky to have her as my Ph.D. advisor. With her enduring support, professional guidance, patient encouragement, and demand for excellence, I have seen my maturation, growth, and achievements increase every day in the past few years. I clearly remember when I first met her on August 17, 2009. It was very difficult for me to communicate with her about research, and I did not know how to do academic research at all. Now, I am very confident about how to solve a challenging new research problem and how to write a paper independently.

As Rome was not built in one day, Prof. Shobha Vasudevan devoted a ton of time and energy to my growth in the past four years. Whenever I was contented with the current result and wanted to stop, she was always there, pushing me to quest for perfection ceaselessly. Her attitude of striving for excellence made my Ph.D. research have greater impact in both academia and industry. Whenever I was not confident about solving a challenging problem, her encouraging eye contact always told me “You can, Lingyi.” Her encouragement gave me confidence to surpass myself every time. Whenever I made a mistake, she did not blame me. Instead, she was always patient and gave me sufficient freedom and time to learn from my own mistake. In that way, I have grown up from every mistake I made.

Her dedication was not limited to my Ph.D. research, but also extended to my daily life. Whenever I had any difficulty or problem in my life, she was there and ready to give me suggestions. She is more like a best friend to me and my wife, instead of just an academic advisor. Thank you, Madam, for your generous help.
During my Ph.D. study, Dr. Xiaotao Chen, Fang Yu, and Cicerone Mihalache from Huawei Technologies collaborated with me on one of my research projects. I would like to thank them for providing two excellent summer internship opportunities and for proposing exciting problems for me to explore. Their timely feedback and constructive comments inspired me frequently and also made my research more practical and meaningful to industry.

I also would like to thank my distinguished Ph.D. committee members Prof. Jiawei Han, Prof. Wen-Mei Hwu, and Prof. Rob Rutenbar for their insightful comments and suggestions on my research work. Many past and current labmates, including David Sheridan, Viraj Athavale, Jayanand Asok Kumar, Sam Hertz, Adel Ahmadyan, Chen-Hsuan Lin, Vinit Shah, Debjit Pal, Sai Ma, Tian Xia, and Parth Sagdeo, have been very helpful in many different ways. I thank all of them.

Last but not least, I would like to thank my parents, my parents-in-law, and my wife, Emily Gong. Without their continuous and selfless love, encouragement, and support, I would never have been able to finish my Ph.D. study. Special thanks to my wife for her faith in me and for giving me freedom to pursue the life I want.
# TABLE OF CONTENTS

**LIST OF TABLES** ............................................. x

**LIST OF FIGURES** ........................................... xiii

**CHAPTER 1**  INTRODUCTION  .................................. 1
  1.1 Hardware Design Methodology: A Glimpse  .................. 1
  1.2 Understanding Hardware Verification  ...................... 2
  1.3 Motivations .............................................. 6
  1.4 Approaches of the Thesis: Static Analysis and Data Mining  8
  1.5 Contributions ............................................. 10
  1.6 Thesis Outline ............................................ 13

**CHAPTER 2**  BACKGROUND .................................... 14
  2.1 Static Analysis of RTL Source Code  ...................... 14
  2.2 GoldMine for Automatic Assertion Generation ............. 16
  2.3 Transaction Level Models  ................................ 19

**CHAPTER 3**  EFFICIENT VALIDATION INPUT GENERATION IN
RTL BY HYBRIDIZED SOURCE CODE ANALYSIS .................. 21
  3.1 Introduction .............................................. 21
  3.2 Positioning of Our Work .................................. 26
  3.3 STAR: Generating Input Vectors for Design Validation by Static
      Analysis of RTL ........................................... 28
  3.4 Path Explosion in STAR .................................... 32
  3.5 Method I: Branch Coverage Guided Input Generation Approach
      (HYBRO) to Attack Path Explosion in STAR ................. 32
  3.6 Experimental Evaluation of Method I  ...................... 37
  3.7 Method II: Symbolic State Caching to Attack Path Explosion in
      STAR ...................................................... 39
  3.8 Experimental Evaluation of Method II  ...................... 45
  3.9 Summary .................................................. 50
# Table of Contents

## CHAPTER 4  TOWARD COVERAGE CLOSURE: USING GOLDMINE ASSERTIONS FOR GENERATING DESIGN VALIDATION STIMULUS

4.1  Introduction ................................................. 51
4.2  Counterexample-Based Incremental Decision Trees .......... 53
4.3  Algorithm Completeness and Convergence Analysis .......... 57
4.4  Coverage Analysis ........................................... 60
4.5  Example: Two Port Arbiter ................................ 66
4.6  Experimental Results ....................................... 70
4.7  Practical Limitation to Achieve 100% Coverage ............. 75
4.8  Discussion about Final Decision Tree ....................... 76
4.9  Related Work .............................................. 80
4.10 Conclusions .................................................... 81

## CHAPTER 5  WORD LEVEL FEATURE DISCOVERY TO ENHANCE QUALITY OF ASSERTION MINING

5.1  Introduction .................................................. 82
5.2  Background .................................................. 84
5.3  A Motivating Example ...................................... 86
5.4  Our Procedure for Automatic Word Level Assertion Generation .... 87
5.5  Simulation Guided Weakest Precondition Computation to Discover Word Level Features ................. 91
5.6  Removing Redundant Propositions .......................... 94
5.7  Experimental Evaluation .................................... 95
5.8  Related Work and Conclusion ................................ 102

## CHAPTER 6  AUTOMATIC GENERATION OF SYSTEM LEVEL ASSERTIONS FROM TRANSACTION LEVEL MODELS

6.1  Introduction .................................................. 103
6.2  Symbolic Execution of TLMs ................................ 107
6.3  TLM Assertion Definition ................................. 108
6.4  Flow of SystemC TLM Assertion Generation ............... 109
6.5  Data Generation ............................................ 111
6.6  Attempt I: Sequential Pattern Mining ...................... 113
6.7  Attempt II: Episode Mining ............................... 115
6.8  Comparison between Sequential Pattern Mining and Episode Mining for TLM Assertion Generation .......... 118
6.9  Quantitative Time Annotation ................................ 118
6.10 Evaluation of TLM Assertions .............................. 119
6.11 TLM Benchmark Platform: An AXI Based Interconnection Network 119
6.12 Experimental Analysis .................................... 120
6.13 Related Work and Conclusion ................................ 129
CHAPTER 7  DIAGNOSING ROOT CAUSES OF SYSTEM LEVEL PERFORMANCE VIOLATIONS

7.1 Introduction ........................................ 132
7.2 Preliminaries ..................................... 135
7.3 Concurrent Pattern Mining Algorithm .............. 138
7.4 Mining Concurrent Patterns for Root Cause Localization .... 140
7.5 A Case Study ........................................ 148
7.6 Related Work ..................................... 153
7.7 Summary .......................................... 154

CHAPTER 8  CONCLUSION .................................. 155

REFERENCES ............................................. 157
LIST OF TABLES

3.1 The coverage, running time, number of patterns and repeated branches reported by HYBRO. .................................................. 36
3.2 Comparison between HYBRO and STAR and HYBRO optimization detail. All runtimes are in seconds. UD chain slicing column represents the percentage of reduced constraint numbers. Local conflict resolution column represents the number of detected conflict when mutating constraint. The speedup column is the running time speedup of HYBRO with two optimizations over HYBRO without two optimizations. The length of each generated patterns is equal to the unrolled cycle number. .... 38
3.3 Comparison between the enhanced STAR introduced in this paper and the original STAR. All runtimes are in seconds. The length of each generated pattern is equal to the unrolled depth. The runtime limit is set as one hour. The original STAR is not scalable for most designs. ......................... 46
3.4 The “UD chain slicing” column represents the percentage of reduced constraint numbers. The “local conflict resolution” column represents the number of conflicts detected during the mutation of constraints. The “caching” column represents the number of detected explored symbolic states. ......................... 47
3.5 The coverage, running time, number of patterns and repeated branches reported by the enhanced STAR. The generated tests by our enhanced STAR have high structural coverage as well as functional coverage. The enhanced STAR is also compared with constraint-based random test generation method. The tests generated by the enhanced STAR have much higher coverage than the tests generated by constraint based random test generation method. ......................... 47
4.1 Coverage of arbiter design ................................. 70
4.2 Improvement on test suites that have high coverage according to standard metrics. The initial test suites have achieved high coverage on some standard metrics. Counterexample based Gold-Mine tests are still able to increase the coverage on other standard metrics. Line, Condition(Cond), Toggle, FSM, and Branch Coverage metrics are shown as standard coverage metrics. 73

4.3 Detecting of injected errors by assertions on OpenRisc module. 75

5.1 Results of our word level feature discovery method. Some bit variables, which are in logic cone but not in predicates, should also be included in features for word level assertion generation. The number of features can be reduced when using word level features. 97

5.2 We show that one word level assertion can cover multiple bit level assertions. We also show the used word level feature for generating word level assertions. 99

5.3 The detecting of injected corner case bugs per word level assertion and bit level assertion. Word level assertions are able to detect more injected bugs. 101

6.1 Evaluation of assertions generated by episode mining for a transaction level model of a DMA controller. Quantitative time constraints are discarded in the assertions since the DMA controller model is a programmer view model, and there is no timing information. 121

6.2 Functional descriptions of the sample set of assertions shown in Table 6.1. Our techniques are able to generate assertions which capture communication specification intent and temporal functionality. 123

6.3 Comparison between episode mining and general sequential pattern mining for TLMs assertions generation on DMA controller model. The number of generated assertions and running time are shown in the table. We also compare the average number of generated TLM assertions per event or function call in the design. 124

6.4 Evaluation the quality of assertions generated by sequential pattern mining. The events within each assertion have no cause-effect relationship, and they are related coincidentally by the sequential mining algorithm. Episode mining, however, is able to avoid the generation of these low quality assertions. 125

6.5 Evaluation of assertions generated by episode mining for an AXI based interconnection network. The unit of time constraint is nanosecond. The used window constraint is 300ns. 126
7.1 Applying domain knowledge I and II to filter the irrelevant transaction traces for mining. The table entries show the retained number of transactions after preprocessing of the transaction traces.

7.2 Sample concurrent patterns discovered using concurrent pattern mining. Ix represents initiator x. Tx represents target x. Bx represents bank x. W represents write operation. R represents read operation. Therefore, (I1, B2, T2) means initiator 1 sends request to bank 2 of target 2.
LIST OF FIGURES

2.1 GoldMine architecture ........................................... 16
2.2 Decision tree building process and assertion generation. ....... 18
2.3 An example about using TLM 2.0 to build system level model. ... 20

3.1 The algorithm flow of STAR. Parameter n specifies the sequentially unrolled depth. ........................................... 29
3.2 An RTL example with instrumented code and its corresponding CFG and expression tree. The broken line indicates the control dependency. ........................................... 30
3.3 HYBRO algorithm flow. The blocks in blue represent the new phases in HYBRO method. .......................... 32
3.4 Branch coverage guided search approach in HYBRO. A comparison to the STAR algorithm is shown. ................. 33
3.5 RTL path enumeration and state space exploration. .......... 40
3.6 Bitmap encoding of symbolic state. S7 is cached as explored symbolic state when path p1 is being explored. S7 is reached again in path p2. ........................................... 44
3.7 The algorithm flow of STAR with explored symbolic state caching. The blocks in blue represent the steps of our explored symbolic state caching method. .......................... 44

4.1 Flow of counterexample-based incremental decision tree algorithm for generating validation stimulus in GoldMine. ........... 54
4.2 Incremental Decision Tree Algorithm. The dotted lines represent parts that are different from GoldMine’s decision tree building approach. ........................................... 54
4.3 Difference between a regular decision tree and an incremental decision tree for an output z and Boolean inputs a, b and c. The counterexample trace is included in the bottom row of the trace data. 55
4.4 The coverage of input patterns in the functional design space for an output. ........................................... 65
4.5 Arbiter: RTL and simulation trace. ................................ 67
4.6 Initial decision tree ........................................... 67
4.7 First iteration: Counterexamples and refined tree ............... 68
4.8 Second iteration: Counterexamples and refined tree ............ 68
A motivating Verilog example [1] for a comparison between word level assertions and bit level assertions. The word level feature and the word level target are highlighted in the word level assertion. Reset signal \( \text{rst} \) is disabled in sample assertions. Mining window length is 2 for temporal assertion generation. The \( \text{Var}(\#) \) in the logic cone denotes the variable’s annotated cycle index.

Our procedure for automatic word level assertion generation. Our contributions, which are shown in dotted block, focus how to automatically discover word level features and targets.

Data structures for weakest precondition computation. The data structures are used for logic cone identification and simulation guided weakest precondition computation. The bold arrow lines show the concrete paths during simulation.

Identification of mutually exclusive features during feature discovery.

The comparison of the number of generated candidate assertions given the same simulation traces. The number of generated candidate assertions is reduced by using word level features.

The comparison of the percentage of true assertions among all candidate assertions. The percentage of true assertions is improved by using word level features.

The comparison of the average number of propositions in true assertions’ antecedent. The fewer number of propositions in antecedent means higher readability.

The increasing of input space coverage with the number of generated word level assertions and bit level assertions. We use the \( \text{alu}_{\text{op}}=\text{OR} \) as target and generate two cycles’ temporal assertions.

A simple program [2] and its corresponding concrete simulation and symbolic execution.

Our vision of SystemC TLM assertion generation. The dotted line outlines the portion of the flow that we have implemented in this chapter. An important use case of our assertions can be as TLM assertions for SystemC model validation and debug or a reference library for RTL assertion generation.
6.3 An example of one simulation run from a timed DMA controller design. The function `dma.write()` is a command called by DMA testbench which configures the controlling register in the DMA controller. `b_transport` is the primitive function call. `mem_read().return` is a function call return.

6.4 A frequent episode example of an event sequence. The window constraint is 3.5 in this example and frequency threshold is 3.

6.5 The incremental candidate episode generation in episode mining. The algorithm incrementally generate candidate episodes with $i + 1$ events from frequent episodes with $i$ events.

6.6 Figure showing the framework of AXI based interconnection network. All interconnection buses are AXI.

6.7 The framework of a transaction level AMBA-based DMA controller.

6.8 Figure showing the distribution of the time interval between two events/function calls of two-event assertions generated by episode mining. We fix one event/function call (write_source_addr) in the DMA controller and consider all assertions including this event.

6.9 The number of TLM assertions and running time for different window constraints. The experimental design is an AXI based interconnection network. As we increase the window constraint further, the number of generated assertions appears to approach that of sequential pattern mining.

7.1 Concurrent pattern in an event sequence and interval window for discovering concurrent patterns.

7.2 Candidate pattern generation. $C_i$ is generated from $L_{i-1}$. $L_i$ is the subset of frequent patterns in $C_i$.

7.3 Transaction trace management using SQL database.

7.4 The flow for root cause localization of performance violation using data mining. The discovered root causes are in the form of generated concurrent patterns.

7.5 Figure showing how domain knowledge II is used to filter irrelevant transaction traces. The red arrow trace from initiator 2 to target 1 shows a latency violated transaction trace. Some operations in the transaction trace are irrelevant to the performance violation.
7.6  Figure showing the relations between concurrent patterns and performance violations. The x-z plane plots the transaction latency versus time, while the x-y plane depicts the occurrences of different patterns at different times, where each frequent occurrence is arranged along the y-axis. Concurrent requests, interleaving read/write accesses, and bank conflict accesses are depicted as color coded triangles or trapezoid.

7.7  Figure showing the number of generated concurrent patterns with and without domain knowledge. The domain knowledge reduces the number of discovered concurrent patterns to less than 10.

7.8  The number of generated concurrent patterns as we increase the size of interval constraint in concurrent pattern mining.
CHAPTER 1
INTRODUCTION

1.1 Hardware Design Methodology: A Glimpse

The advance of process technology along with reductions in the cost of silicon have made it possible to integrate more and more transistors into a single chip of silicon. Our capability in designing such complex chips has not improved over the past years. Hardware design methodology enables us to efficiently design such increasingly complex chips. In this section, we give an overview of the design methodology that has been adopted in industry. This can help the reader appreciate the status and role of hardware verification in chip design flow.

To design a chip, we start with the specification, which includes functionality of the chip, the performance requirement, the power requirement, physical constraints like size and area, and fabrication technology. In this thesis, we are concerned with the functionality and the performance.

At the system level, we build the system level model according to the specification. This system level model is then used to decide what kind of architecture and interconnect is the best for the design. In that process, we need to verify the functionality of the system model against the specification and evaluate the performance of different architectures in different usage scenarios. In some processor based applications, the system level model provides a virtual platform for software development at an early stage.

System level model creates an abstract and high level view of the design. The detailed logic implementation for each block is done at the register transfer level (RTL), in which a synchronous digital circuit is modeled in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals. We use a hardware description language (HDL) like Verilog or VHDL to create an RTL model for the design. In parallel, functional verification ensures that the RTL design satisfies the specification.
Once functional verification of the RTL design has been completed, the RTL design is synthesized into an optimized gate level netlist. Physical constraints such as timing and area are considered in this process. Synthesis tool tries to meet these constraints by calculating the cost of various implementations. Physical implementation converts the netlist into a GDSII layout file, which is finally fabricated in foundry.

1.2 Understanding Hardware Verification

Due to the high fabrication costs of hardware, detecting and preventing errors in hardware is vitally important. Fixing bugs after delivering the chip to customers is difficult, and chip callback will incur considerable loss to companies. In the early 1990s, the notorious Pentium FDIV bug caused Intel a loss of about $475 million replacing faulty processors. Hardware bugs can even be catastrophic in safety-critical systems, such as nuclear power plants and traffic control system. Therefore, hardware verification is critical to the design of reliable hardware.

Verification is a primary source of bottlenecks in the hardware design cycle. Due to the growing complexity of hardware systems, verification has already become the dominant cost in the full design process. According to ITRS [4], verification engineers outnumber design engineers, with this ratio reaching two or three to one for most complex designs. The widely adopted verification technologies can be mainly grouped into two categories: formal and simulation-based verification.

1.2.1 Formal Verification

Static formal verification aims to prove the functional correctness of a design with respect to a mathematical formal specification. The purpose is to establish software or hardware system correctness with mathematic rigor. In general, there are three formal methods used in hardware verification. The first one is based on model checking for finite state machine verification [5]. The second one is deductive verification. The last one is equivalence checking.

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for that model [6]. The properties are expressed in propositional temporal logic [7], and the system is modeled as a state-transition graph. The model checking procedure
efficiently traverses all reachable states of the system and automatically determines whether the property is satisfied by the state-transition graph [8]. In general, there are two possible outcomes: the specified property passes the model checking, or the property is falsified. In the latter case, a counterexample is also generated to explain why the property fails. Users can take advantage of the counterexample to fix the design bug. However, model checking suffers from the state explosion problem, which means that the state space of the system being verified must be relatively small. The increasingly complex hardware is still several orders of magnitude too large to be verified by model checking [9], although there has been a lot of research in combating the state explosion problem, such as binary decision diagram (BDD) based symbolic model checking [5], abstraction [10], bounded model checking [11], and compositional model checking [9].

Interactive theorem proving is a typical deductive verification technique, and it verifies the system by equational reasoning. A collection of mathematical proof obligations are generated from the system and its specification. The truth of these proof obligations implies the correctness of the system [12]. The advantage of interactive theorem proving is that it does not impose a priori limit on the state space of the system that can be verified. The user can control the verification problem size during the verification process by breaking down proofs about very large systems into proofs about small components [9]. The disadvantage is that it requires manual guidance during the verification process. The user needs to provide a sequence of theorems to the verification system.

Equivalence checking verifies whether two descriptions/implementations of a circuit function are equivalent. In other words, the two descriptions should exhibit exactly the same input/output behaviors. Typically, the two descriptions are at different levels of abstraction. For instance, a gate level implementation is checked against the RTL design or an RTL design implementation is checked against the system level model. There are combinational equivalence checking and sequential equivalence checking. In combinational equivalence checking, the two designs are first transformed into a single miter circuit [13], and then the output of the miter circuit should be proved to be constant 0. The proving algorithm can use the Boolean satisfiability engine. Sequential equivalence checking is more complex and requires the state traversal of the product machine of two sequential designs. In practice, structural similarities between two designs are always exploited to reduce the complexity of sequential equivalence checking [14].
1.2.2 Simulation Based Verification

Due to the scalability issues of formal verification, the simulation based verification (also known as dynamic verification) is still the dominant verification method for complex SoC. It simulates the design by providing stimuli on the inputs to exercise the design. The observed responses on the output are checked against the specification. Coverage analysis measures whether the design has been adequately exercised and the functionality of the design has been sufficiently covered.

The stimulus generator is the most challenging part of developing a simulation based verification environment/testbench. In practice, it uses directed tests or constraint-random generated tests. Directed test generation relies on the verification engineers to manually construct certain test cases according to the specification. In constraint-random method, verification engineers provide a set of constraints on the design inputs according to design specification. A constraint solver then randomly generates tests satisfying the constraints. Although directed tests capture much of the desired system behavior, they do not suffice in checking for unintentional erroneous behavior. The constraint-random method is intended to capture infrequent or unexpected design behavior.

In order to evaluate the comprehensiveness of the simulated tests and the degree to which the design has been simulated, coverage metric is utilized in simulation to provide a quantitative measurement. Multiple types of coverage metric are available in: code coverage, structural coverage, and functional coverage. Coverage feedback information can be used to guide further test generation [15].

1.2.3 Assertions in Verification

Assertions or invariants [16] provide a mechanism to express desirable or required properties that should be true in the system. Assertions accurately encode intended behavior in the form of temporal logical expressions for a state transition system. The design can then be verified against those assertions.

Assertions are used for validating hardware designs at different stages through their life cycle, such as pre-silicon formal verification, dynamic validation, runtime monitoring, and emulation [17]–[19]. Assertions are also synthesized into hardware for post-silicon debug and validation and in-field diagnosis [17]. In formal verification, assertions can serve as the formal specification (properties) that must be satisfied on the design. A model checking tool verifies whether the assertions...
are true or false on the design.

In simulation-based verification, those executable assertions are employed to monitor the dynamic simulation, improve the internal design signal observability, and reduce the debug effort [18]. First, the assertion monitors dynamically check whether the simulation conforms to all provided assertions. In addition, traditional simulation-based verification checks the input and output behaviors against specification. It is a black-box test and lacks internal signal observability. Assertions provide the internal test points and turn the black-box simulation into a white-box simulation. Finally, verification engineers typically have to trace back to internal signals tediously when there is a mismatch on outputs. The adoption of assertions simplifies that diagnosis and detection of bugs, since the failed assertions pinpoint the location of an error.

1.2.4 System Level Verification

The design of SoCs is becoming increasingly complex with the deployment of multi-core processors and embedded memories. ESL design and verification is an emerging electronic design methodology that focuses on the higher abstraction level concerns first and foremost. ESL verification involves the tasks of checking the functionality and evaluating the performance and power at the early stage of system development. Raising the abstraction level results in faster development of prototypes, simulation, and earlier system validation and software development. From the perspective of verification, the reduction of implementation details in system level design can increase the simulation speed and allow for a more global view of the complex system. Transaction level designs, also called transaction level models (TLMs), are widely used for ESL modeling. The structural data transmitted between modules is abstracted to a transaction. Details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture. TLMs speed up simulation and allow for exploring and validating design alternatives at a higher level of abstraction [20].
1.3 Motivations

1.3.1 Simulation Test Generation

Because of the infeasibility of exhaustive simulation, the termination point of random simulation is very nebulous. Contemporary industries often use a numeric value, like a few million simulation cycles, before concluding the random simulation phase. Evidently, such a methodology is unsystematic and inconclusive. Despite various coverage metrics used to evaluate the simulated tests, there is no assurance that there are no gaping holes in the design behavior. Coverage closure, or the process of determining the completeness of functional coverage of input vectors, is one of the most daunting challenges of the present-day validation environment. It is desirable to have a scalable and systematic input vector generation strategy towards coverage closure.

1.3.2 Assertion Generation

Generating good assertions poses a major challenge for design verification. Assertions in current verification flow are manually written by verification engineers. Manual construction of minimal, but high functional coverage assertions takes multiple iterations and man-months to achieve. Too many assertions can result in degradation of simulation performance. Too few assertions, however, cannot guarantee sufficient functional coverage. In sequential designs, temporal assertions spanning several cycles are usually the source of subtle, but serious bugs. Writing temporal assertions is also more challenging than writing combinational assertions.

In the software community, the assertions/invariants are mainly used to improve maintainability and readability of programs. For example, invariants can be inlined in software releases to prevent bugs. Any future modifications to the software should not break the expected invariants. Program verification also relies on invariants to prove program correctness [21]–[23]. The Floyd-Hoare approach requires that the program loop exhibits a loop invariant, also known as an inductive assertion. The loop invariant should be preserved during every execution of the loop body. However, it is difficult for programmers to write these invariants. The software engineering community has exerted a great deal of effort towards building automatic invariant generators to make the creation of invariants less manpower
intensive.

Abstract interpretation and the constraint-based approach are the two most widespread frameworks for static invariant inference [24]. Abstract interpretation performs an approximate symbolic execution of programs until an assertion is reached that remains unchanged by further executions of the program [25]. Constraint-based techniques use decision procedures over non-trivial mathematical domains (such as polynomials or convex polyhedra) to represent concisely the semantics of loops with respect to certain template properties [26].

Dynamic invariant inference for software was pioneered by the Daikon tool [27] and has gained significant attention in the software engineering community. In a nutshell, the Daikon approach tests a large number of candidate properties against several program runs. The properties that are not violated in any of the runs are retained as likely invariants, which implies that the inferred invariants are not sound.

In the hardware community, academia and industry have recently proposed several solutions [28]–[35] to automate the assertion generation process. GoldMine [29] is a tool specifically designed for automatic assertion generation from RTL design. The engine combines data mining and static analysis techniques. It mines the simulation traces of a behavioral RTL design using a decision tree based learning algorithm to produce candidate assertions. Note that the candidate assertions may be spurious due to the causality established by simulation data. They are passed to a formal verification engine to filter out spurious assertions.

Current assertion generation solutions like GoldMine generate assertions at the bit level, and term-level information from the RTL abstractions is completely lost. Even if there are word level variables in RTL, all bits are ungrouped. Those mechanically generated bit level assertions have low readability and are typically not in a human-digestible form. Frequently, designers find the machine generated data too difficult to parse and assimilate since it is at a lower level of abstraction. In addition, each generated bit level assertion has very low coverage of the input space, and the bit level assertions tend to be repetitive, and therefore numerous. These disadvantages drastically limit the usability of the mechanically generated bit level assertions.
1.3.3 System Level Verification

The state-of-the-art of modeling and verification of ESL designs is still unsystematic and very ad hoc. The commercial ESL tools provide very limited support to ESL verification [36]. To improve the observability of the entire system, system engineers always instrument code in ESL models to collect dynamic simulation information. This instrumentation requires users to be familiar with the models. Also, due to fast simulation speed, the collected execution trace files are so large that it becomes awkward to analyze them manually and further debug the ESL models.

Lifting the assertion based verification methodology to the system level is one promising way to improve the efficiency of ESL verification. Assertions can be adopted for on-line monitoring of the model simulation or serve as a formal specification of ESL models. Assertions were recently introduced for the verification of SystemC designs [37], [38], but there is no automatic assertion generation method. Another challenge is that all previous definitions of system level assertion do not take into account the performance specification/intent of ESL models. It is indispensable to have an automatic method for the generation of system level assertions, which express both functionality and performance.

When using TLMs for complex SoC design, system engineers can simulate the models and then evaluate the SoC performance through an off-line analysis of all transaction traces. However, current tools are not able to automatically analyze and troubleshoot the root cause of a performance violation from an enormous amount of transaction trace data. According to our industrial collaborator, localizing the root cause of a tough performance violation could take system engineers two days to two weeks, which tremendously increases the time-to-market of their product.

1.4 Approaches of the Thesis: Static Analysis and Data Mining

In this thesis, the combination of high level static analysis and data mining based dynamic analysis is extensively explored for hardware verification. Although the individual techniques depend on the target application domain, all of them adhere to this principle.

Static analysis literally refers to the techniques used to reason with all possible behaviors of a system without executing the system. In hardware, we use the term
static analysis to mean methods that analyze design structure/function (analogous to program syntax and semantics in software) [39]. Examples of structural methods include cone-of-influence [40], localization reduction [41], and weakest precondition computation [24]. Formal verification and symbolic execution [42] are also considered forms of static analysis of the semantics of a model.

Data mining is the process of deciphering knowledge from data [3]. Data mining uses dynamic behavior in the form of simulation data or training sets to find statistical correlations and make inferences about a system. Typically, data mining finds its applications in web-mining, online recommendation systems, health-care, and bio-informatics. In data mining community, a set of mining algorithms has been proposed for different application problems, such as association rule mining, decision tree based learning, and sequential pattern mining [3].

1.4.1 Why Data Mining for Verification?

Simulation based verification is still the main method for verifying complex SoC. During simulation-based verification, tremendous amounts of data are generated due to the fast simulation speed in both RTL and ESL. Performing detailed analysis of the simulation data to hunt design bugs is like finding a needle in a haystack. Data mining naturally lends itself to our verification problem and is an ideal technique for performing the simulation data analysis. We expect that data mining will replace manual waveform analysis, and the knowledge discovered from the simulation data could ease hardware verification.

1.4.2 Data Mining Guided by Static Analysis

Simply application of data mining on the dynamic simulation data from hardware has several disadvantages, which can be offset by static analysis. First, data mining is not able to simulate judgment, i.e., it will not be able to decide how interesting a piece of information (say, a rule) is to the hardware verification domain [3], [43]. Without domain knowledge, data mining algorithm tends to produce a lot of uninteresting information to verification engineers. Static analysis is able to extract domain knowledge from the designs. For example, data mining is used to derive logic rules between two signals in the design. Static analysis techniques such as cone of influence can identify whether the two signals influence each other. If the
two signals are independent, any logic rule between the two signals is spurious and cannot be used for hardware verification.

Second, static analysis can aid the feature extraction of data mining. In the raw simulation data, all high level design variables are ungrouped into bit level signals. If we use the bit level signals as features in mining algorithm, the discovered knowledge is hard for the verification engineer to digest since high level design intents are lost. Our static analysis technique can extract high level structures from the design and retain them as features for data mining.

Third, static analysis is useful for preprocessing raw simulation data and thus guiding the mining engine to generate more meaningful patterns for verification engineer. In constraint-random verification, some of the design inputs are randomized in the stimulus generator. As a result, it is difficult for the mining engine to identify potential relationship between signals. For example, signal $a$ and signal $b$ have the potential relationship $a = b$. In random simulation, they are concretized to $a = 1$, $b = 1$ in one simulation trace, and $a = 2$, $b = 2$ in another simulation trace. A data mining algorithm would not be able to find the relationship $a = b$. However, our static analysis technique can transform the simulation data and assign a symbolic value to $a$ and $b$, which helps the mining engine discover the relationship $a = b$.

1.5 Contributions

1.5.1 RTL Verification

**Systematic and Scalable Input Vector Generation through Hybrid Analysis of RTL Source Code**

We propose HYBRO and symbolic state caching methods for generating high coverage input patterns in RTL [44],[45]. The two methods both use a hybrid approach that combines dynamic and static analysis of the RTL source code. HYBRO improves the scalability of input vector generation by considering branch coverage as the metric for guiding the input vector generation. We implement the Verilog RTL symbolic execution engine and show that the notion of branch coverage helps alleviate the inefficiencies caused by previous path-based approaches to input vector generation and also achieve high coverage. We also describe two types of optimizations, dynamic slicing and local conflict resolution, that increase the efficiency of
HYBRO significantly.

We also propose an explored symbolic state caching method to attack path explosion. Explored symbolic states are states starting from which all subpaths have been explored. Each explored symbolic state is stored in the form of bitmap encoding of branches to ease comparison. When the explored symbolic state is reached again in the following symbolic execution, all subpaths can be pruned. The symbolic state caching method is promising in showing high coverage on benchmark RTL designs, and the runtime of the test generation process is reduced from several hours to less than 20 minutes.

**Stimulus Generation for Coverage Closure Using GoldMine Assertions**

We propose a methodology for attaining coverage closure of design validation using GoldMine spurious assertions [46], [47]. We take advantage of the counterexamples from the formal engine to incrementally generate stimulus and finally achieve coverage closure. GoldMine uses a formal verification tool to check the generated assertions and a counterexample is generated for each spurious assertion. In our methodology, we feed these counterexample traces to the simulation engine to iteratively refine the original simulation trace data. We introduce an incremental decision tree to mine the new traces in each iteration. The algorithm converges when all the candidate assertions are true. Our algorithm will always converge and capture the complete functionality of each output of a sequential design on convergence. Our method always results in a monotonic increase in simulation coverage. We also introduce an output-centric notion of coverage, and argue that we can attain coverage closure with respect to this notion of coverage.

**Word Level Feature Discovery for Enhancing RTL Assertion Mining**

We propose a technique that uses static and dynamic analysis of RTL code to discover word level features [48]. We use simulation guided weakest precondition computation to discover word level features in terms of primary inputs. A post processing of assertions is employed to remove redundant propositions. The generated word level features are used by machine learning algorithm. This allows the generated assertions to be at the same level of abstraction as RTL. We do not modify the learning algorithms themselves to achieve our goal. The machine learning algorithm, as such, is agnostic to the level of abstraction of its features. By
using those discovered word level features, the machine learning algorithm is able to generate the word level assertions. Experimental results on Ethernet MAC, I²C, and OpenRISC designs show that the generated word level assertions have higher expressiveness and readability than their corresponding bit level assertions.

1.5.2 ESL Verification

Assertion Mining for System Level Design

We propose a method for automatically generating assertions from Transaction Level Model (TLM) simulation traces [35], [49]. The generated assertions express design specifications in the form of linear temporal logic with quantitative temporal constraints [50]. We first generate the assertions without regard to the quantitative time constraints. They are mined in the form of frequent patterns in the simulation traces. We mine simulation traces using episode mining to identify frequent episodes comprising function calls and events. We then annotate the episodes with real time parameters to express quantitative time constraints among the function calls or events in the episode. When mining such TLM assertions, we employ symbolic execution to generalize the parameters and return values of function calls in the traces to help the mining engine generate high quality assertions. We have constructed a realistic AXI-based interconnection network platform on which we demonstrate experimental results. We show that our technique efficiently generates high quality performance and functional assertions on the AXI-based platform and a transaction level AMBA-based DMA controller. We demonstrate that episode mining is more scalable and able to generate a more compact set of high quality TLM assertions than previous efforts using sequential pattern mining. The number of generated assertions using episode mining can be reduced by up to 228 times, and the time interval between two events/function calls in each assertion is smaller than 50 time units.

Diagnosing Root Causes of System Level Performance Violations

We propose a methodology to localize root causes of latency or throughput violations [51]. We use a concurrent pattern mining approach to infer frequent patterns from transaction traces to localize root causes. We apply three categories of do-
main knowledge from the violations and models to filter the irrelevant transaction traces and increase the effectiveness of the mining results. We provide three culprit scenarios to the mining algorithm by including transaction traces relevant to the corresponding culprit scenario. The mined concurrent patterns then belong to that culprit scenario. We provide a case study for diagnosing performance violations of an experimental platform and show that our domain knowledge can reduce the number of transaction traces up to 92.8%. The concurrent pattern mining pinpoints the root cause to one of fewer than $10$ patterns among $100000$ transaction traces.

1.6 Thesis Outline

The remainder of this thesis is organized as follows.

In Chapter 2, we present the background on the techniques used in the thesis.
In Chapter 3, we first present our previous work STAR for RTL validation input generation and show the path explosion problem in STAR. We then present HYBRO and symbolic state caching methods to attack the path explosion problem and thus improve the efficiency and scalability of the input generation method.
In Chapter 4, we present our methodology to generate input stimuli to achieve coverage closure using GoldMine.
In Chapter 5, we present the methodology to discover word level features using static and dynamic analysis of the RTL source code.
In Chapter 6, we present our technique for generating assertions from system level designs using GoldMine. We compare two different mining algorithms for system level assertion generation.
In Chapter 7, we present our methodology for localizing root causes of latency or throughput violations using a concurrent pattern mining approach.
In Chapter 8, we present a summary of the work and conclude this thesis.
CHAPTER 2
BACKGROUND

In this chapter, we present some definitions and background on the techniques used in the thesis.

2.1 Static Analysis of RTL Source Code

2.1.1 Basic Definition

We treat the RTL source code as a “program” as in [39], [40]. We analyze the CFG of the RTL design.

A simple path in the RTL CFG is a path which is executed in a single cycle. A sequential path refers to a path that is executed across multiple time cycles. In order to account for the sequential behavior of the RTL, the RTL CFG is sequentially unrolled. This means that the CFG is replicated many times, with the variables in each unroll being annotated by the corresponding relative time cycle.

In the CFG, a conditional node \( N_i \) is control dependent on conditional node \( N_j \) if the outcome of evaluating \( N_j \) determines if \( N_i \) should execute or not. \( N_j \) is said to dominate \( N_i \). A control dependency graph is a data structure that maintains the control dependencies within a single time cycle.

2.1.2 Symbolic Execution

Symbolic execution [52] refers to the execution of a single path with symbolic inputs. Symbolic execution of a path generates symbolic expressions that are a logical conjunction of the guards and assignments to the variables used in guards along that path. Symbolic execution of a sequential path generates expressions such that every variable along the path is annotated by the time cycle to which it belongs. A
constraint in our context is a symbolic expression translated in a form that is accept-
able by an SMT constraint solver. The symbolic execution described here considers
only the synthesizable subset of Verilog, which make the design of execution en-
gine easier and the generated symbolic expression compatible with SMT solver.
The entire execution engine is working on the CFG and expression tree structure of
each statement.

Symbolic execution follows only the current concrete simulation path instead of
considering all control flow paths in a process. This can dynamically reduce the
state space considered to a large extent when compared to symbolic simulation. In
addition, the symbolic simulator handles all paths in the control flow graph, some
of which actually are infeasible paths due to the conditional dependency in different
behavioral design blocks. However, the symbolic execution engine always follows
the feasible path whose feasibility has been demonstrated by concrete simulation.
Our symbolic execution engine is integrated with a commercial simulator, from
which the dynamic concrete execution path information is collected.

2.1.3 Weakest Precondition Computation

In software verification, Dijkstra’s weakest precondition [23] of a program state-
ment $S$ is a function mapping any postcondition $R$ to a precondition, which is de-
noted as $wp(S, R)$. It is the weakest precondition on the initial state ensuring that
execution of $S$ terminates in a final state satisfying $R$.

In case of hardware RTL, the assignments to all state variables happening in one
cycle correspond to state transitions in hardware. Such state transition relations can
be viewed as the statement $st$ in $wp(st, P)$ [53].

Let $I$ be the set of primary input variables and let $\bar{r} = < r_1, r_2, ... r_n >$ be the
set of all state variables. We denote the transition function for state variable $r'_i$ as
$r'_i = T_i(I, \bar{r})$. The transition relation is $\bar{T} = < T_1, T_2, ... T_n >$. We let $E(\bar{r'})[\bar{r} \setminus \bar{T}]$
denote the substitution of each $r'_i$ appearing in expression $E$ with its corresponding
transition function $T_i$. The weakest precondition of predicate $P$ with respect to
transition relation is defined as follows:

$$wp(\bar{T}, P(\bar{r'}, I')) = P(\bar{r'}, I')[\bar{r} \setminus \bar{T}]$$

1. $I'$ and $r'$ represents the primary inputs and register variables in next cycle. $r$ and $I$ are in current
cycle. $r, I$ and $I'$ may coexist in the resulting $wp$. 

15
The resulting weakest precondition is still a predicate. Therefore, we can compute the weakest precondition backward for \( k \) consecutive cycles. We denote it as \( wp^k(T, P) \). Specifically, we use \( wp^0 \) to denote the weakest precondition in a single cycle if the variables used in \( P \) are intermediate wire variables. Because these intermediate variables are neither primary inputs nor register variables in the RTL, \( wp^0 \) computes the weakest predicates on primary inputs in a single cycle. We henceforth omit \( T \) in the \( wp \) notation.

2.2 GoldMine for Automatic Assertion Generation

GoldMine combines two diverse technologies, data mining and static analysis, to generate assertions automatically. Data mining comprises dynamic, statistical techniques that are computationally efficient, but depend heavily on domain information for deriving relevant knowledge from a system. Static analysis of designs (including formal verification) captures domain (design) information, but suffers from computational complexity issues. Together, these two technologies offset each other’s disadvantages. The static analysis, when used to guide the data mining, gives rise to useful domain knowledge, i.e. assertions. Figure 2.1 shows the architecture of GoldMine. It is composed of a Data generator, Static analyzer, A-Miner, Formal verifier and A-Val components.

![Figure 2.1: GoldMine architecture](image)

In GoldMine, the generated assertion form is \( G(a \Rightarrow b) \), \(^2\) where \( a \) is propositional and \( b \) can be temporal. We are only considering bounded assertions, so we allow “next” operator(\( X \)). The specified bound is given by the mining window length. It is not possible for us to generate unbounded liveness properties expressed using “eventually” operator \( F \). Although “until” operator \( U \) has not been shown in the examples of this chapter, some of our assertions can be interpreted as a “bounded”

\(^2\)We use LTL [7] notation for expressing GoldMine assertions. We can produce SVA [54] as well as PSL assertions.
until operation. For example, \( a \land Xa \land XXa \Rightarrow XXXb \), we can potentially extend this to a contrived “until” proposition like \( aUb \). It should be noted that it is not faithful to the true semantics of the Until operation, since the observations are only within a mining window.

2.2.1 Data Generator

The data generator phase of GoldMine is used to provide the data for the data mining algorithm. For a given RTL design, the data is obtained through dynamic simulation traces. The design is simulated for a fixed number of cycles (10,000) using random input patterns. Regression tests, if available, can also be applied to obtain traces.

The sequential behavior of a design is usually expressed in the form of temporal assertions. GoldMine is capable of generating combinational as well as sequential assertions. We need to provide a mining window length, or the duration of time cycles for which we want to capture temporal behavior. For instance, if we want to consider the following behavior: once \( a \) is valid, \( d \) will be valid two cycles later, the mining window length can be set to 2. The generated assertions will span up to a maximum of 2 cycles. For example, \( a \Rightarrow XXd \) can be one form of generated assertions.

For sequential behavior, the sequential variables within mining window are annotated with the cycle in which they are assigned. Then the sequential variables with cycle annotation are treated the same manner as combinational variables. In this example, variable \( a \) can be annotated as \( a(t - 1) \), \( a(t) \) and \( a(t + 1) \) within mining window.

2.2.2 Static Analyzer

The static analyzer extracts domain-specific information about the design and passes it to the data mining algorithm. We extract the logic cone of influence for every output that we are interested in assertions for. The data mining phase (A-miner) is restricted to analyzing only the logic cone of any output [29]. This limits the search space of the mining algorithm from all the inputs, to the relevant variables in the design.
2.2.3 Decision Tree Based Learning and A-Miner

The data mining phase of GoldMine is called A-miner, for assertion mining. A-Miner uses a decision tree based supervised learning algorithm to map the simulation trace data into conclusions or inferences about the design.

In the decision tree, the data space is locally divided into a sequence of recursive splits on the input variables. Each decision node implements a “splitting function” with discrete outcomes labeling the branches. This hierarchical decision process that divides input data space into local regions continues recursively until it reaches a leaf. We require only Boolean splits at every decision node, since our domain of interest is digital hardware. The example in Figure 2.2 for an output $z$ shows the simulation trace data for inputs $a$, $b$ and $c$.

An error function picks the best splitting variable by computing the variance between target output values and the values predicted by decision variables. The predicted value on each node is the mean of output values, denoted by $M$, while the error at a node is denoted by $E$ in the example. When the error value becomes zero, it means all output values are identical to the predicted value and the decision tree exits after reaching such a leaf node. When the error value is not zero, the variable with minimum error value is chosen to form the next level of decision tree. A candidate assertion is a Boolean propositional logic statement computed by following the path from the root to the leaf of the tree. In the example, the splitting of input space into two groups after decision on variable $a$ leads to $E = 0$, corresponding to assertion $A_1$. Along the $a = 1$ branch, another split occurs on $b$. Assertions $A_2$ and $A_3$ are obtained at the leaf nodes.

Figure 2.2: Decision tree building process and assertion generation.
2.2.4 Formal Verifier and A-Val

The candidate assertions inferred by A-miner are based purely on statistical correlation metrics like mean and error. We restrict the candidate assertions we consider to those with 100% confidence. This means that even if a single example in the trace data does not subscribe to a rule generated by the tree, the rule will be discarded. Despite this strict restriction, A-miner may still infer candidate assertions that are true of the simulation data, but are not true of all possible inputs. To identify candidate assertions that are system invariants, the design as well as the candidate assertions are passed to a formal verification engine. If a candidate assertion passes the formal check, it is a system invariant. Otherwise, the formal verifier generates a counterexample trace that shows a violation of the candidate assertion. The SMV [5] model checking engine is a part of GoldMine, along with a commercial model checker. In the example in Figure 2.2, $A_1$ is declared false, while $A_2$ and $A_3$ are declared true. In GoldMine, A-val forms the evaluation phase for the assertions, to bridge the gap between the human and machine generated assertions.

GoldMine provides a radical, but powerful, validation method. Through mining the simulation trace, it reports its findings in a human digestible form (assertion) early on and with minimal manual effort. However, in GoldMine, there is no concept of feedback from any phase to the data miner. Given that data mining performs very effectively when given feedback, we have incorporated feedback from the formal verification phase for enhancing the simulation test data in Chapter 4.

2.3 Transaction Level Models

Transaction level models (TLMs), also called transaction level designs, separate the details of communication among computation modules from the functional details of these computation modules [20]. The communication is through channels. The channel’s interfaces provide a set of communication primitives (function calls) to computation modules and hide low-level communication protocol details. Computation modules are connected via their ports to the channels. Transaction level modeling in SystemC involves communication between SystemC processes using function calls. The designers then fully focus more on communication between the processes in computation modules instead the algorithms performed by the processes themselves. Transaction level designs are typically employed for perfor-
mance evaluation of different architectures, software development or as reference models for RTL designs. Additionally, this high level model greatly improves simulation performance and helps to shorten the time to market.

In practice, designers adopt different TLM coding style and model accuracy when modeling a hardware design for different purpose. TLM can be untimed or timed model [20]. Untimed TLM is mainly used for software programmers in early software development and is also called programmer’s view (PV). Timed TLM is appropriate for the use cases of architectural exploration and performance analysis.

TLM 2.0 is the latest industrial standard for transaction level modeling from the OSCI [55]. The standard allows for model interoperability throughout the design community. In TLM 2.0 library, a transaction is a data structure(class) communicated between modules using function calls. An initiator module is responsible for initiating a transaction and a target module responds to transactions initiated by other modules. The same module like a router can be both an initiator or a target. An example system level model using TLM 2.0 can be shown in Figure 2.3. Module A serves as an initiator and calls the library function nb.transport_fw of socket1 to transmit a transaction to target Module B through forward channel. Target Module B gives response through backward channel.

![Figure 2.3: An example about using TLM 2.0 to build system level model.](image-url)
CHAPTER 3

EFFICIENT VALIDATION INPUT GENERATION IN RTL BY HYBRIDIZED SOURCE CODE ANALYSIS

3.1 Introduction

The validation phase of RTL design is widely accepted as being responsible for perpetual bottlenecks in the design cycle. It is estimated that over 70% of design time and resources is spent on design validation [4]. In current practice, application of known stimuli or directed tests helps capture expected behavior of the design. Although development of directed tests is an arduous task involving many man-months, the directed test suites usually converge at an acceptable point. However, in the case of random stimuli, such confidence is far from being achieved. Even in state-of-the-art industrial environments with many dedicated validation resources, the design is considered as stable after the application of a large number (>1 trillion) of random patterns. Since that metric is devoid of information regarding design behavior coverage, it is very unsatisfactory.

In this chapter, we first introduce STAR (STatic Analysis of RTL), a technique for automatic generation of high coverage functional vectors in RTL using static analysis of the HDL source code. This technique directly manipulates and analyzes RTL source code as a “program” instead of reasoning with logic gates. STAR specifically uses symbolic execution [52] of the RTL in conjunction with concrete simulation (a.k.a. concrete execution) to form a practically feasible, efficient input vector generation strategy.

Symbolic execution in RTL is adapted from that in software [52]. It refers to the execution of a single program path with symbolic inputs. As a result of symbolic execution, the symbolic path constraint for that path is generated. In RTL, the design is simulated using symbolic values of inputs, instead of concrete values. Symbolic execution is different from symbolic simulation, which has been applied widely at the gate level [56] and at the RTL [39]. Symbolic simulation is a static methodology that deciphers all possible executions by traversing RTL source code.
Instead, symbolic execution follows a single concrete execution path of RTL and symbolizes it. Since it is dynamic, it only considers feasible paths, a luxury that a static engine like a symbolic simulator does not have. A test is generated using constraint solving of the symbolic constraints. The STAR algorithm divides the RTL design into feasible paths and generates tests to cover each path. This “divide-and-conquer” strategy can be employed for a combinational (or a single cycle) design as well as a sequential design. For sequential designs, the RTL code is unrolled for a number of cycles, and an RTL path can be across several cycles.

The STAR algorithm consists of several basic steps. Initially, a random concrete stimulus is applied to the design, and an execution path is obtained. The RTL expressions in the concrete simulation path are extracted using symbolic execution. The expressions are composed of branch conditions as well as assignment statements that use RTL operators. We refer to the branch condition as guard and the expressions as symbolic expressions. The conjunction of guards in terms of input variables provides the path constraints under which the concrete path is executed. One or more of these constraints are now inverted (or mutated). The resulting symbolic expression now corresponds to another path in the design. The Satisfiability Modulo Theories (SMT) constraint solver [57], which is a SAT-based decision procedure for linear arithmetic logic, is used to solve the new path constraints and produce an input vector pattern that is a test for the new path. The inversion of constraints can be done systematically to cover all paths in a region in a depth-first manner or a breadth-first manner.

STAR generates input vector patterns for all paths of an RTL design. For a combinational design, these are simple paths. For a sequential design, these are sequential paths that involve sequential unrolling over multiple cycles. The semantics of sequential “always” process denotes a process that can potentially loop forever. As a result, a huge number of paths must be enumerated. We unroll a sequential RTL design for as many cycles as required to completely describe the temporal behavior of the variables, or the sequential depth. Within the unrolled RTL design, a sequential path starts at the initial cycle and ends at the unrolled cycle. Given any intermediate cycle $k$, a sequential path is divided into two subpaths: one from the initial cycle to cycle $k$, and another from cycle $k$ to the end of the unrolled cycle. In STAR, our algorithm involves the mutation of guards on a sequential path, and each mutation generates a new path. This leads to path explosion [58], a situation in which the number of paths will increase exponentially with the number of branches in each cycle and the unrolled cycles, i.e., unrolling depth. Such exhaustive enumeration
of all possible paths limits the scalability of the STAR approach severely, making it effective only for small designs with few paths.

To attack the path explosion problem, we present HYBRO (HYbrid analysis and BRanch Coverage Optimizations), a methodology to generate input vectors in RTL automatically and with extremely high coverage. HYBRO circumvents the path explosion problem faced by STAR by considering branch coverage as the metric for guiding the path exploration. In STAR, the extracted symbolic expressions are placed onto a constraint stack, containing the guards of a single symbolic execution. In order to systematically explore the design, one of the guards is mutated or inverted. The STAR algorithm terminates when all the conditional expressions in the RTL or guards in the constraint stack have been exhausted. This exhaustive enumeration of all paths leads to the path explosion problem. In contrast, HYBRO uses a coverage driven approach to mutate a guard and give a symbolic expression to the SMT solver. In HYBRO, the instrumented code is also used to record branch coverage in the RTL CFG. At the stage when a guard is picked for mutation, if all the branches in the CFG that depend on the mutated guard have already been covered, a different guard is picked from the symbolic expression. The process terminates when there are no more guards that have not been mutated in the constraint stack. HYBRO uses the guidance provided by branch coverage in eliminating repetition of paths. This makes the analysis much more efficient as compared to STAR.

Additionally, in this chapter, we also present two optimizations that increase the efficiency of HYBRO. Both of them draw on the static analysis and dynamic analysis technology. The first optimization is dynamic UD(Use-Definition) chain slicing. The UD chain is a data structure consisting of a use(U) of a variable and all definitions(D) of the variable that can reach the use without any other intervening definitions. This approach removes redundant constraint in the path constraints. The second optimization involves resolving local conflicts when making guard mutation. The successful detection of conflict can reduces calls to the SMT solver [57].

Although HYBRO tries to exhaustively stimulate all reachable branches in the CFG, it does not guarantee complete coverage. It can be viewed as a best effort process that practically produces excellent coverage. An important advantage of HYBRO is that it produces controllability in the input vector generation process, allowing the process to be guided to uncovered regions of the design. Once inside a region, HYBRO will explore many branches in the region.

Our experimental results show high structural coverage as well as functional coverage within reasonable time as compared to the STAR algorithm. Additionally, the
two optimizations can speed up the HYBRO by 1.6-12 times on various benchmarks.

To attack the path explosion problem and also sustain the algorithm completeness of STAR, we present symbolic state caching solution to ameliorate the path explosion problems faced by STAR, thereby scaling this technology further. The unrolling of the “always” processes, despite having an infinite cycle of evaluations, has finite number of reachable states. In the concrete execution phase, the reached state in a cycle along the concrete path is represented as a set of values of state (register) variables in that cycle. When we symbolically execute such a concrete path, we obtain the reached symbolic state in the corresponding cycle. Such a symbolic state is represented as constraints on the register variables. These constraints are in terms of primary input variables in previous cycles. When considering a sequential path for test generation, we analyze the reached symbolic state in every cycle that the path goes across. If a path (subpath) reaches a symbolic state that has already been reached, then we do not need to generate tests for that path (subpath).

We outline how we enumerate paths and analyze their corresponding symbolic states. Any enumerated sequential path starts from an initial cycle and extends up to the sequential depth. A symbolic state is reached in every cycle. STAR follows a depth-first order to mutate guards. In other words, a guard is mutated from the last unrolled cycle to the initial cycle. When mutating a guard between cycle \( k \) and the last unrolled cycle, the subpath from the initial cycle to cycle \( k \) as well as the reached symbolic state at cycle \( k \) are kept unchanged. Only when all subpaths after cycle \( k \) have been enumerated can the guards in cycle \( k \) be mutated. In this situation, we consider the reached symbolic state at cycle \( k \) to be the explored symbolic state, which means that all subpaths following this symbolic state have been enumerated before.

The explored symbolic state at cycle \( k \) is now recorded. When enumerating a new path, we check the reached symbolic state in every cycle against the previously recorded explored symbolic states. If a reached symbolic state is identical to any explored symbolic state and the cycle \( r \) of the reached symbolic state is not less than the cycle parameter \( k \) of the explored symbolic state, all subpaths starting from this reached symbolic state will not reach any new symbolic state and can be pruned. If \( r < k \), the subpath starting from cycle \( r \) will be longer than the subpath starting from cycle \( k \), and a new symbolic state can possibly be reached. We specifically refer to this recording method as explored symbolic state caching. It can help to prune paths and thus circumvent the path explosion problem.
The challenging parts of the *explored symbolic state caching method* pertain to (a) representation of the symbolic states and (b) comparison of two symbolic states. If the symbolic states are represented as extracted constraints, an SMT solver can be used to compare two symbolic states. As mentioned earlier, a subpath from the initial cycle to cycle $k$ reaches a symbolic state at cycle $k$. In other words, this subpath can determine the symbolic state. The *taken branches* on the subpath starting from initial cycle can uniquely represent that subpath. We can then use the set of taken branches on the subpath to represent the corresponding symbolic state. However, not all taken branches on the subpath are needed to represent that symbolic state. If the definition in a taken branch are not used in the symbolic state constraint, the taken branch is not used to represent the symbolic state. We use a backward tracing method to identify all necessary taken branches. After we have determined the set of taken branches to represent a symbolic state, the comparison of two symbolic states is reduced to comparison of two sets of taken branches.

In this chapter, we further propose an optimization that will increase the performance efficiency and reduce the memory consumption of comparing two sets of taken branches. The branches in a cycle are organized as one *bitmap* of that cycle, where one bit corresponds to one branch. If the branch is used to represent symbolic state, the corresponding bit is set. Finally, a reached symbolic state in cycle $k$ is represented as a set of bitmaps in several continuous cycles ending at cycle $k$. If two sets of bitmaps are identical, their represented symbolic states will be the same.

In our implementation, we also apply the two optimizations from HYRRO, which further improve the efficiency of constraint solving.

Our experimental results show a significant improvement over the original STAR algorithm. We report high structural coverage as well as functional coverage within reasonable time, as compared to STAR. The caching of explored symbolic states can effectively avoid the exploration of repetitive subpaths. The number of repetitively covered branches is less than 6% of that in original STAR. Also, our method is able to detect up to 11850 explored symbolic states on different benchmark designs. We also show that the tests generated using our method have much higher coverage than those generated using constraint random test generation method.
3.2 Positioning of Our Work

We discuss work that is related to different aspects of our explored symbolic state caching method from the perspectives of both hardware validation and software testing. Despite the differences in semantics and in actual results, software and hardware techniques for verification, test, and validation have often been mutually inspired [53], [59]–[61] demonstrate the application of predicate abstraction from software in hardware designs, while [5], [62] present successful hardware techniques that have been used to inspire software model checking. We then describe the challenges in applying an idea that has worked in software testing in our context.

Our work is mainly inspired from software concolic testing. We treat hardware RTL design as a software program as in [39],[40]. This is different from traditional testing or simulation techniques in hardware that view RTL as an abstraction of gate level semantics, and therefore view RTL constructs as aggregates of bit level operations. We view an RTL object as a software language construct, facilitating static analysis of the source code without giving it a gate/transistor level interpretation until we actually need to. To the best of our knowledge, the integration of the explored symbolic state caching method and symbolic execution in the context of design validation has not been presented before.

3.2.1 Hardware Validation

In hardware, hybrid techniques that combine dynamic (simulation) with static analysis at the gate level have been used for formal verification [63]. [64]–[66] use cost functions derived from static abstraction to guide random simulation.

Static analysis of RTL has been used for verification [39],[40],[53],[67] and manufacturing level testing [68]. Symbolic simulation technology was initially used at the gate level for formal verification [56],[69]. Recently, some researchers tried to develop the RTL symbolic simulator [56],[70],[71]. However, our symbolic execution engine considers one feasible path each time, instead of the entire design, and is more scalable to large designs. There has also been some prior work leveraging the static structure of RTL to speed up model checking [72],[73].

Coverage-guided or specification-driven stimulus generation [15],[74]–[77] and development of effective coverage metrics [78],[79] have been looked at in depth before. For microprocessor verification, a graph-theoretic model is developed in
to capture the structure and behavior of pipelined processors, and test programs are generated to detect function faults in the model. Test generation in RTL targeting at stuck-at manufacturing faults has been explored to reduce test generation time [81].

### 3.2.2 Software Concolic Testing

In software testing, extensive research has been done on the idea of combining concrete execution and symbolic execution to automatically generate tests [82]–[84]. Both concolic testing [83] and dynamic test generation [58] belong to this category. Many tools based on symbolic execution have been developed to automatically generate tests. These tools are very promising for finding bugs in real-world software [82], [85]–[88].

### 3.2.3 Attacking Path Explosion

The problem of path explosion of symbolic execution in software testing has also been approached in a variety of ways. One method uses heuristics to guide path enumeration to hit uncovered statements in the code at the expense of completeness [82], [89]. Another method, RWset [90], prunes redundant paths by tracking the memory locations read and written by the checked code. The work in [58],[91],[92] uses compositional methods to reduce the number of paths to be solved by the constraint solver. State merging is employed in [93] to improve the performance of symbolic execution. Our approach distinguishes itself from these work in two respects. First, our symbolic state caching approach is being done in the new context of hardware RTL design validation. The HDL used to describe RTL designs is semantically different from software languages. Second, we propose the use of bitmap encoding of branches to represent symbolic states to attack path explosion.

### 3.2.4 Challenges of Applying Concolic Testing to RTL Designs

Our idea was inspired by concolic testing in software testing. Adapting it to RTL design validation has several challenges, since the semantics of HDLs used to describe hardware RTL designs are different from that of a sequential software language.
HDLs model the sequential design as multiple always/process blocks in Verilog and VHDL. A clock signal is used to trigger these blocks in a synchronous design. Non-blocking assignments to register variables in the current cycle take effect in the next cycle.

Also, the multiple always/process blocks in RTL design are executed in a concurrent, non-deterministic manner during simulation. Different interleaving execution orders of the blocks during simulation could non-deterministically produce different results.

Finally, HDLs model non-terminating, reactive systems. An always/process block is an endless loop in simulation. Unlike sequential software, no path is a simple path. The number of paths will increase exponentially as the number of unrolled cycles increases. From that perspective, the path explosion problem in hardware RTL design is even more severe than that in software.

Hence, the adaption is conceptually nontrivial. In fact, the adaption is orthogonal to software techniques, since the expected improvements and results are totally different. Hardware is a finite-state system, and many paths repetitively cover the same symbolic state.

### 3.3 STAR: Generating Input Vectors for Design Validation by Static Analysis of RTL

In this section, we first introduce the data structure used in STAR, and then briefly explain each step of STAR. Figure 3.1 shows the algorithmic flow of STAR.

Figure 3.2 (a) shows a Verilog RTL example design with instrumented code. For each single statement or conditional expression in the design, the expression tree structure exactly records the corresponding assignment or expressions for later constraint generation and is linked to corresponding CFG node. As shown in Figure 3.2, the conditional expression in line 11 is represented in the expression tree linked to the branch node. All the nonblocking statements in line 24 are represented in the expression tree structures that are linked to the corresponding CFG node i5. The expression tree can also be used to build the use-define chain for the design since it is easy to deduce the used variable and defined variable from expression tree. For example, in a non-assignment expression, all leaf node variables are the used variables in the expression.
3.3.1 Code Instrumentation and Static Analysis

As shown in diamond in CFG in Figure 3.2 (b), all the instrumented branch variables keep track of the concrete simulation path at each cycle by sustaining an array which is indexed by cycle number. At the end of each cycle, the instrumented branch variables are compared with their value in the last cycle. The updated variable means the corresponding branch is taken, which is recorded in corresponding element in the array. When the concrete simulation is done, the symbolic execution can exactly follow the executed concrete path.

The instrumentation process is automatically done by a Verilog parser. The RTL design is directly instrumented with source code that is meant to trace a concrete execution path. In Figure 3.2, the instrumented code has been underlined. The values of $i_0$ will change if $reset$ evaluates to 1 during simulation, and the value of $i_1$ will change if $reset$ evaluates to 0 during simulation. A change in the value of either $i_0$ or $i_1$ indicates which branch was executed by the concrete simulation.

After the construction of the CFG and expression tree for the given RTL, the CFG and expression tree are statically analyzed to obtain the UD chain. The UD chain is mainly used to determine whether a variable is used in a branch condition expression.
3.3.2 Concrete Execution for Multiple Cycles and Recording of Concrete Path

A concrete input pattern is given as a stimulus. The first time, this stimulus is generated at random. For every subsequent iteration of the algorithm, the concrete input patterns are generated by the SMT solver automatically. The concrete stimulus is applied on the design. If any input variable does not get an assignment by SMT solver, a random value will be used in simulation. During the concrete simulation, the CFG branch node of the instrumented code records the concrete path.

3.3.3 Symbolic Execution to Extract Path Constraints

In this step, the concrete path recorded in the control flow graph is symbolically executed to generate path constraints. The symbolic execution engine walks the CFG in the design one by one at every cycle. In each CFG, it only follows the concrete simulation path. At each branch node in each cycle, the engine decides the taken path by looking up the corresponding element in the array. At each node in the path, the corresponding expression tree is traversed and output as symbolic expression.
As shown in Figure 3.2 (b), the engine arrives at the CFG branch node \( if(c>d+e) \) in cycle \( i \) and traversing of linked expression tree can generate the following constraint: \( c[i]>d[i]+e[i] \).

For nonblocking assignments to register variables in the path, the assigned register variable will take effect in next cycle. Therefore, the cycle index should be set to the next cycle number. Taking the assignment \( out<e\&f \) in node \( i5 \) for example, the generated constraint will be \( out[i+1]=e[i]\&f[i] \). Finally, the conjunction of all generated symbolic expressions forms the corresponding path constraints.

### 3.3.4 Constraint Mutation

After the generation of one path constraint, we need to mutate the guard to enumerate another path. In order to systematically enumerate all possible paths, we use a depth-first order to mutate guards. A *constraint stack* stores all extracted constraints according to CFG traversal order. It should be noted that a guard will not be considered as candidate guard if it is mutated last time. A mutate flag is used to indicate whether a guard has been mutated before. When a guard is popped out from the constraint stack, its mutate flag is reset. When a guard is mutated, its mutate flag is set. This flag can guarantee that the same path cannot be enumerated twice. When there is no candidate guard, all possible paths are enumerated, and STAR exits.

### 3.3.5 Constraint Solving and Next Pattern Generation

The mutated constraint is passed through an SMT solver. If a satisfiable assignment is generated, it will be used as the next concrete input pattern. If the SMT solver reports an unsatisfiable result, it means that the mutation leads to an infeasible path. A new guard in the constraint stack is chosen for mutation. If some inputs have been removed from the stack in the guard mutation phase, they will not be a part of the constraint. They might need to be randomly generated in the next concrete pattern. However, the random generation, along with the existing constraints, will direct the entire test generation into another *region* of the CFG. The entire algorithm will be repeated for the new region.
3.4 Path Explosion in STAR

The symbolic execution engine enumerates all possible sequential paths in the design to generate tests for sequential behavior. However, the number of sequential paths in an RTL design will increase exponentially with the number of branches in the RTL code and the sequentially unrolled depth. Each guard mutation in any cycle will lead to a new sequential path of the RTL design. In the RTL example shown in Figure 3.2, there are 4 paths in each cycle, and there will be $4^5$ possible paths if the sequentially unrolled depth is 5. Because of path explosion, it is difficult to apply STAR for large designs.

In this chapter, we present two different methods to attack the path explosion issue and thus improve the scalability of STAR. The first one is HYBRO, which uses branch coverage to guide the constraint mutation. HYBRO only mutates the guard leading to the uncovered area in each iteration. HYBRO is an incomplete method and cannot guarantee to cover all paths of RTL design. The second one is symbolic state caching method. It caches the previously explored symbolic state and try to avoid the exploration of repetitive state space in future.

3.5 Method I: Branch Coverage Guided Input Generation Approach (HYBRO) to Attack Path Explosion in STAR

![HYBRO algorithm flow](image)

Figure 3.3: HYBRO algorithm flow. The blocks in blue represent the new phases in HYBRO method.
Figure 3.3 shows the algorithmic flow of HYBRO. We describe the new phases of the algorithm over STAR.

\[ c[t+1] \leq d[t+1] + e[t+1] \]

if (reset)

\[ i_0 \]

\[ i_1 \]

case (state)

\[ i_2 \]

\[ i_5 \]

exit

cycle \( t \) cycle \( t+1 \)

Constraint stack after extraction

\[ c[t+1] = d[t] \]

out\( [t+1] = d[t] \& c[t] \]

state\( [t+1] = s_1 \)

\[ c[t+1] > d[t+1] + e[t+1] \]

\[ c[t+1] = e[t] \]

out\( [t+2] = c[t+1] \& d[t+1] \)

state\( [t+2] = s_2 \)

reset\( [t+1] = 0 \)

reset\( [t] = 1 \)

HYBRO method

retained

removed

removed due to UD chain slicing

If (c > d + e)

Figure 3.4: Branch coverage guided search approach in HYBRO. A comparison to the STAR algorithm is shown.

3.5.1 Recording Branch Coverage

A concrete input pattern is given as stimulus. For the first time, this stimulus is generated at random. For every subsequent iteration of the algorithm, the concrete input patterns are generated automatically. The concrete stimulus is applied for a predetermined number of cycles. Every time a branch executes in the concrete input simulation, the edges corresponding to the instrumented code in the CFG are marked as covered. In the concrete execution shown in Figure 3.4, the corresponding concrete pattern is \( \text{reset}[t] = 1, d[t] = 1, e[t] = 1, f[t] = 0; \text{reset}[t + 1] = 0, d[t + 1] = 0, e[t + 1] = 0, f[t + 1] = 1 \). The bold edges correspond to the concrete execution path in the CFG. In the first cycle \( \text{reset} = 1 \) is applied, so the edge leading to \( i_0 \) is marked as covered. This is shown by the large dot on the arrows. In the second cycle \( \text{reset} = 0 \) is applied, and the branches leading to \( i_1, i_2 \) and \( i_3 \) are marked as covered. The branch leading to node \( i_4 \) is marked as covered from a previous iteration of the algorithm.

33
3.5.2 Dynamic UD Chain Slicing in Path Constraint Extraction

In this step, the concrete path identified in the control flow graph is symbolically executed. For the concrete path executed over multiple cycles, the corresponding symbolic execution will also involve variables across multiple time cycles. For the example concrete path, the symbolic execution will yield the following expression.

In cycle $t$: $reset[t] = 1 \land c[t + 1] = d[t] \land out[t + 1] = d[t] \land \& c[t] \land state[t + 1] = s1$.

In cycle $t + 1$: $reset[t + 1] = 0 \land state[t + 1] = s1 \land c[t + 1] > d[t + 1] + e[t + 1] \land c[t + 2] = e[t + 1] \land out[t + 2] = c[t + 1] \land \& d[t + 1] \land state[t + 2] = s2$. The $state[t + 1] = s1$ appearing in cycle $t$ corresponds to the non-blocking assignment in line 5 in Figure 3.2 (a). Its appearing in cycle $t + 1$ corresponds to the guard in line 9 in Figure 3.2 (a).

The regular constraint extraction mechanism would be to simply reuse the symbolic expression. However, we introduce an optimization strategy here that makes use of the UD chain. We traverse the CFG from the last time cycle backwards to the first time cycle in current dynamic execution to apply this optimization. For every variable in every guard in a cycle we refer the UD chain to see where it was defined. Among all possible definitions for a used guard variable found by static analysis, we only consider the one that has been executed by the concrete input vector. We mark all the definitions transitively from the last cycle to the first cycle. At the end of this analysis, if a definition has been marked, it must be required by a guard in a subsequent frame. Otherwise, it is discarded from the constraint.

The constraint is extracted into the constraint stack such that every element of the stack corresponds to a term that is a conjunct in the symbolic expression. For example, in Figure 3.4 the constraint is pushed into the constraint stack such that each element is annotated with the cycle number and the lowest cycle number is at the bottom of the stack. Since UD chain slicing analyzes the CFG from last cycle to first cycle, the constraint stack elements need to be popped and then re-pushed into the stack. The UD chain slicing optimization is intended to make the size of the constraint smaller. In Figure 3.2 (a), there are four definitions in the constraint in cycle $t$ for the used variable $c$ in line 11 in cycle $t + 1$. They are $c <= d$ in line 4, $c <= e$ in line 14, $c <= f$ in line 19 and an implicit definition $c <= c$ in i5. However, only $c <= d$ in line 4 in cycle $t$ is extracted as a constraint since it is in the concrete path. In addition, the definition $out[t + 1] = d[t] \& c[t]$ can be removed from the constraint since variable $out$ is not used in the following cycle.
3.5.3 Branch Coverage Guided Constraint Mutation

A constraint is said to be mutated if any of its guards is inverted/mutated. In this step, the guard at the top of the constraint stack is selected as a candidate for mutation. The candidate guard is mutated and then analyzed using the CFG that was marked by branch coverage as follows. If the mutated candidate guard $g$ corresponds to a control node in the control dependency graph, and all the branches leading to the nodes that are control dependent on $g$ have already been covered, then $g$ is discarded from the constraint stack. If any of the nodes dependent on $g$ can be executed by branches that have not yet been covered, $g$ is retained. Intuitively, if $B$ is control dependent on $A$, it means that some path in the program that goes through $A$ can bypass $B$, and $A$ is the point in which this divergence can occur. So, it suffices to look at the control node that dominates the other nodes for doing a branch overage analysis. The branch coverage analysis is performed for only one cycle at a time. The cycle that is considered for a guard corresponds to the annotated in the guard variable. So, in the example, only the $(t+1)^{th}$ unroll is analyzed for control dependency and branch coverage.

In Figure 3.4, the guard $c[t+1]>d[t+1]+e[t+1]$ is the mutation candidate. Definitions/Assignments in the constraint are not considered mutation candidates. The guards in the constraint are shown by the shaded elements of the stack. These will be mutation candidates. The candidate guard is mutated to $c[t+1]\leq d[t+1]+e[t+1]$. The mutated guard now corresponds to the node $i4$. We first check if $i4$ dominates other uncovered conditional nodes (including itself). However, $i4$ is marked as covered in previous iteration of the algorithm. So this guard is removed from the constraint in our approach. In future cycles ($t+2$ and beyond) of the algorithm, $i4$ might dominate other control nodes according to the control dependency graph in Figure 3.2 (b). If there are control dependent nodes that are not yet covered, the current mutated candidate guard will be retained.

As shown in Figure 3.4, the STAR algorithm would have retained this guard, irrespective of it being along a branch that has been covered. This would result in repetitive coverage of the paths that execute control nodes that are dependent on $i4$ in all future iterations as well. Our approach manages to avoid repetitive path traversal for input pattern generation by using the notion of branch coverage.
Table 3.1: The coverage, running time, number of patterns and repeated branches reported by HYBRO.

<table>
<thead>
<tr>
<th>Bench-mark</th>
<th>Cycles</th>
<th>Bran_Cov</th>
<th>Path_Cov</th>
<th>Assert_Cov</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>b01</td>
<td>10</td>
<td>94.44%</td>
<td>94.44%</td>
<td>95%</td>
<td>0.07s</td>
</tr>
<tr>
<td>b06</td>
<td>10</td>
<td>94.12%</td>
<td>93.10%</td>
<td>100%</td>
<td>0.10s</td>
</tr>
<tr>
<td>b10</td>
<td>10</td>
<td>87.10%</td>
<td>72.73%</td>
<td>4.71%</td>
<td>4.56s</td>
</tr>
<tr>
<td>b10</td>
<td>30</td>
<td>96.77%</td>
<td>81.82%</td>
<td>68.58%</td>
<td>52.14s</td>
</tr>
<tr>
<td>b10</td>
<td>50</td>
<td>96.77%</td>
<td>81.82%</td>
<td>93.65%</td>
<td>180.42s</td>
</tr>
<tr>
<td>b11</td>
<td>10</td>
<td>78.26%</td>
<td>78.26%</td>
<td>43.97%</td>
<td>0.28s</td>
</tr>
<tr>
<td>b11</td>
<td>50</td>
<td>91.30%</td>
<td>91.30%</td>
<td>100%</td>
<td>326.85s</td>
</tr>
<tr>
<td>b14</td>
<td>15</td>
<td>83.59%</td>
<td>13.36%</td>
<td>100%</td>
<td>301.69s</td>
</tr>
<tr>
<td>or1200-0</td>
<td>50</td>
<td>93.75%</td>
<td>77.78%</td>
<td>100%</td>
<td>191.82s</td>
</tr>
<tr>
<td>or1200-1</td>
<td>100</td>
<td>93.75%</td>
<td>77.78%</td>
<td>100%</td>
<td>191.82s</td>
</tr>
<tr>
<td>or1200-1</td>
<td>50</td>
<td>96.30%</td>
<td>79.07%</td>
<td>94.12%</td>
<td>219.00s</td>
</tr>
<tr>
<td>or1200-2</td>
<td>100</td>
<td>96.30%</td>
<td>79.07%</td>
<td>100%</td>
<td>92.15s</td>
</tr>
<tr>
<td>or1200-3</td>
<td>5</td>
<td>91.53%</td>
<td>90.20%</td>
<td>96.67%</td>
<td>19.07s</td>
</tr>
<tr>
<td>or1200-3</td>
<td>10</td>
<td>96.61%</td>
<td>96.08%</td>
<td>100%</td>
<td>297.62s</td>
</tr>
</tbody>
</table>

3.5.4 Local Conflict Resolution

There are two kinds of local conflict when making guard mutation. First, the same guard occurs across multiple processes in the same cycle in RTL. If this guard is part of the constraint stack and gets mutated, this will result in a conflict with the same guard that is present lower in the stack. Similarly, if the previous definition(s) of a used variable in a guard are assigned a constant value, the mutation of that guard to another value will give rise to a conflict. We detect syntactically equivalent guards shared across multiple processes in a single cycle when doing static analysis. If such a shared guard is a candidate for mutation, it is directly popped out from the stack. Another local conflict occurs when all used variables in current mutated guard are assigned a constant value in the variable’s definition of current path. For example, before the mutation of guard \(a>b\), we first trace the definition of \(a\) and \(b\) through UD chain. If both of the definitions assign a constant value to \(a\) and \(b\), the mutation of \(a>b\) will definitely lead to a local conflict. This case often takes place for case statement in the design. In the example shown in Figure 3.4, if the guard variable state is a candidate for mutation, we can pop out the guard from the constraint stack since state is assigned a constant in its definition. These local conflicts are supposed to be detected in the SMT solver phase. However, as an optimization, we detect such conflicts before we pass the constraint to the SMT solver.
3.6 Experimental Evaluation of Method I

We have implemented the HYBRO algorithm and all optimization strategies with C++, which interact with VCS simulator through the direct programming interface (DPI) and Yices [57] constraint solver with its C Library Interface. All the following experiments are performed on a four Intel i5 2.67GHz processor cores machine with 16GB of memory running Linux. We present a set of experimental results on some examples of RTL model from ITC99 and OpenRISC1200 [1]. Four OR1200-x designs are instruction cache controller, data cache controller, Wishbone bus interface and exception handling logic.

3.6.1 Structural Coverage Evaluation

The first experiment in Table 3.1 shows the coverage rate for the generated test patterns using HYBRO. It can be observed that HYBRO can achieve very high structural coverage as long as the unrolled cycle number is enough. For most of these designs, all the feasible branches in the design are fully covered even if the tool does not report 100% coverage due to the infeasible paths. For example, there may be unreachable `default` branch for `case` statement in a design.

The unrolled cycle number is an important parameter to improve the coverage in HYBRO. This parameter is determined by the coverage feedback. If the coverage is not high, it means the uncovered branches are not reachable in the unrolled cycles. We can increase the unrolled cycle number. The $b_{10}$ and $b_{11}$ circuit demonstrate this relationship between coverage and unrolled cycle number. When the unrolled cycle number increases from 10 to 30, all the feasible branches are fully covered. However, for or1200-2 and or1200-3 designs, 10 cycles is enough to cover all branches. The running time exhibits the applicability of HYBRO for practical circuit. There is no memory bottleneck since HYBRO does not store any states of the circuit.

The only exceptionable design is $b_{14}$ circuit. In this design, several uncovered branch conditions in the design depend on the overflow of a big counters. As a result, it becomes difficult to satisfy these branch conditions.

A very interesting benefit from HYBRO is that it can identify and report infeasible paths. This is highly valuable to the verification engineer. In addition, HYBRO can also be used to check properties on each path.
Table 3.2: Comparison between HYBRO and STAR and HYBRO optimization detail. All runtimes are in seconds. UD chain slicing column represents the percentage of reduced constraint numbers. Local conflict resolution column represents the number of detected conflict when mutating constraint. The speedup column is the running time speedup of HYBRO with two optimizations over HYBRO without two optimizations. The length of each generated patterns is equal to the unrolled cycle number.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Cycles</th>
<th>STAR</th>
<th>HYBRO</th>
<th>HYBRO Optimization detail</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Runtime</td>
</tr>
<tr>
<td>b01</td>
<td>10</td>
<td>1.64s</td>
<td>1024</td>
<td>0.07s</td>
</tr>
<tr>
<td>b10</td>
<td>15</td>
<td>&gt;3600s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>b11</td>
<td>15</td>
<td>293.80s</td>
<td>84342</td>
<td>111736</td>
</tr>
<tr>
<td>or1200-0</td>
<td>10</td>
<td>&gt;3600s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>or1200-3</td>
<td>10</td>
<td>&gt;3600s</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### 3.6.2 Functional Coverage Analysis

We also demonstrate that the input patterns generated by HYBRO can provide high functional coverage although the entire analysis in HYBRO is on RTL structure. The RTL code structures reflect the design specification and HYBRO is able to follow these structures to generate meaningful patterns from the perspective of interesting functionality.

We use the assertion tool GoldMine [29] to generate assertions for several designs. The generated tests are also applied on the design to evaluate the assertion coverage through simulation. The assertion is normally used to check whether the design has correctly implemented the specification. The triggered assertion coverage rate actually reflects the quality of simulation patterns. The assertion coverage report of our generated test patterns by HYBRO is shown in the assertion coverage column of Table 3.1, from which we can conclude that HYBRO is able to comprehensively capture the design function.

Comparing with random method, HYBRO has the advantage to generate all meaningful input to cover all possible design behavior instead of exhausting all value of input. The generate process relies on the hint from RTL structure. Comparing with widely used constraint-random generation method, HYBRO doesn’t have to manually build the constraints.
3.6.3 Optimization Effect

The second experiment shown in Table 3.2 is a comparison between HYBRO and the STAR algorithm [44]. We set the maximum runtime to one hour. It can be observed that STAR suffers from path explosion for most of the designs. This is shown by high runtime and large number of patterns. The runtime will increase exponentially as the unrolled cycle number increases in STAR. Therefore, the unrolled cycle number in this experiment is very small when applying STAR. It is unscalable for big practical design. For the same circuit, however, the runtime for HYBRO is very small. The runtime of HYBRO does not increase exponentially with the unrolling cycle number. We can see that the number of repeated branches is very high in STAR, whereas in HYBRO, the average number of repetitively covered branch is less than 5% of that in STAR.

The optimization detail column in Table 3.2 shows the optimization effectiveness of UD chain slicing and local conflict resolution. UD chain slicing reduces the number of constraints sent to SMT solver since unused variable definition will be excluded from current constraints. Local conflict resolution reduces the number of calling to SMT solver since it relies on the static analysis information to check the conflict in constraints stack instead of sending them to SMT solver. We specifically run HYBRO with the two optimizations and without the two optimizations. It can be observed that the optimization can speed up HYBRO by 1.6-12 times on various circuits.

3.7 Method II: Symbolic State Caching to Attack Path Explosion in STAR

3.7.1 Path Enumeration and Reachable State Space

The execution of a simple RTL path in each cycle leads to the design’s transition from one state to another. A sequential path will generate continuous state transitions. In every cycle, different states are reached by following different sequential paths starting from the initial cycle, and we say these states are reached states in that cycle. In the example shown in Figure 3.5, the letters A–O represent simple paths of each cycle. There are two possible reached states in cycle $t - 3$: $S1$ and $S2$. The simple paths in cycle $t - 2$ can generate the following state transitions:
$S_1 \rightarrow S_3$, $S_1 \rightarrow S_4$, $S_1 \rightarrow S_5$, $S_2 \rightarrow S_5$, and $S_2 \rightarrow S_6$. Figure 3.5 also shows two sequential paths, $p_1$ and $p_2$, and their corresponding state transitions along the paths.

In STAR, we unroll the RTL design for a specified number of cycles and enumerate all feasible sequential paths for test generation. By using the constraint stack, we keep track of each guard and cycle number to avoid the enumeration of repetitive paths. In Figure 3.5, provided that the unrolled depth is $t + 1$ and the current state at cycle $t - 3$ is $S_1$, we will enumerate all sequential paths according to the following order:

1. $\cdots A \rightarrow D \rightarrow G \rightarrow I$;
2. $\cdots A \rightarrow D \rightarrow G \rightarrow J$;
3. $\cdots A \rightarrow D \rightarrow H \rightarrow K$;
4. $\cdots A \rightarrow D \rightarrow H \rightarrow L$;
5. $\cdots B \rightarrow E \rightarrow G \rightarrow I$;
6. $\cdots B \rightarrow E \rightarrow G \rightarrow J$;
7. $\cdots B \rightarrow E \rightarrow H \rightarrow K$;
8. $\cdots B \rightarrow E \rightarrow H \rightarrow L$;
9. $\cdots C \rightarrow F \rightarrow G \rightarrow I$;
10. $\cdots C \rightarrow F \rightarrow G \rightarrow J$;
The “···” in the above sequential paths represents the common subpath between the initial cycle and cycle \( t - 1 \). STAR combines the concrete execution and symbolic execution to generate input patterns for each sequential path. During concrete execution of each such sequential path, the reached states in the path are explicit states, since all registers are evaluated to concrete values. During symbolic execution of the sequential path, the states reached in each cycle are represented as reached symbolic state. Each symbolic state represents a set of concrete states satisfying the symbolic state constraints. A reached symbolic state has a cycle parameter specifying the cycle where the symbolic state is reached.

### 3.7.2 Explored Symbolic States

Although STAR does not repetitively enumerate the same sequential path, the same symbolic state may be reached again and again by different sequential paths. In the example shown in Figure 3.5, the symbolic state \( S7 \) at cycle \( t - 1 \) is reached by three different sequential paths: \( A \to D \), \( B \to E \), and \( C \to F \). When STAR explores sequential path \( \cdots A \to D \cdots \), all subpaths following the symbolic state \( S7 \) will be enumerated. State \( S7 \) is actually identified as explored symbolic state in cycle \( t - 1 \) when “\( \cdots A \to D \to G \to I \)”,”\( \cdots A \to D \to G \to J \)”,”\( \cdots A \to D \to H \to K \)”, and “\( \cdots A \to D \to H \to L \)” are all enumerated for test generation. When subpath \( \cdots B \to E \) is being followed, all the same subpaths from symbolic state \( S7 \) are enumerated again.

Given that a previously explored symbolic state in cycle \( k \) is reached again in cycle \( r (r \geq k) \) through a different sequential path, it is not necessary to enumerate all subpaths following that explored symbolic state. In the example shown in Figure 3.5, when sequential path \( B \to E \) is being enumerated after guard mutation, it is no longer necessary to enumerate all subpaths starting from symbolic state \( S7 \). It should be noted that \( r \) should be no less than \( k \). In the case that \( r < k \), the subpaths from cycle \( r \) to the end of the unrolling cycle will be longer than the subpaths from cycle \( k \) to the end of the unrolling cycle, and new states may be reached. Therefore, all subpaths following the explored symbolic state in cycle \( r \) are still required to be enumerated for test generation. In the example shown in Figure 3.5, the unrolled depth is \( t + 1 \). The state \( S7 \) is reached in cycle \( t - 1 \) and \( t \). Assuming that we first
identified \( S7 \) as explored symbolic state at cycle \( t \), there are two possible transitions in cycle \( t + 1 \): \( S7 \rightarrow S9 \) and \( S7 \rightarrow S10 \), the \( k \) is equal to \( t \). It will not reach \( S11, S12, \) and \( S13 \) since the unrolled depth is \( t + 1 \). Now if the \( S7 \) is reached again in cycle \( t - 1 \) later and \( r \) is equal to \( t - 1 \), it does not satisfy \( r \geq k \). We should continue to explore all possible subpaths starting from \( S7 \) at cycle \( t - 1 \). We can then reach \( S11, S12, \) and \( S13 \), which are not reached before.

Our main idea is to record/cache the explored symbolic state during path enumeration. If all guards after the current cycle are popped out from the constraint stack, all subpaths following the reached symbolic state in the current cycle have been explored. We can use symbolic execution to generate the constraints for explored symbolic state and reached symbolic state. When a new path is enumerated, the reached symbolic state in every cycle is checked against the previously recorded explored symbolic states. If the state in the current cycle is an explored symbolic state, all constraints after the current cycle in the constraint stack are popped out and will not be mutated for generating a new path. In other words, all subpaths following the explored symbolic state are pruned.

The pruning of subpaths has no impact on functional coverage of RTL design. All pruned subpaths are redundant from the perspective of validation test generation. Given an explored symbolic state \( S \) at cycle \( i \), all subpaths starting from symbolic state \( S \) at cycle \( i \) to the unrolled cycle are explored. In other words, we have generated all stimuli to explore these subpaths before. When the state \( S \) is reached again, we no longer need to generate such stimuli. Even if we input the symbolic constraints of these subpaths to SMT solver, SMT solve still generates exactly the same assignments as before for these subpaths.

### 3.7.3 Caching Explored Symbolic State

The symbolic execution engine that we have built can be used for calculating symbolic constraints for reached symbolic states in the concrete path. Note that explored symbolic states also belong to reached symbolic states. Given a cycle number and a concrete sequential path, the symbolic execution engine is able to extract constraints for the reached state in any given cycle. These constraints are in terms of primary inputs in previous cycles. However, there are two problems if we represent explored symbolic state symbolically:

1. How can we efficiently cache the explored symbolic state?
2. How can we efficiently compare a reached symbolic state with the cached and explored symbolic state?

Directly recording all constraints for a symbolic state is not memory-efficient, and it is difficult to compare the constraints of two symbolic states. An SMT solver can be used to compare two symbolic states. However, it is highly inefficient, since each pair of states will call the SMT solver once. We propose to use a set of taken branches to represent the symbolic state. When extracting constraints for symbolic state, symbolic execution traverses the CFG only along the concrete path. The taken branches on the concrete subpath suffice to determine the generated constraints for the reached symbolic state. Intuitively, we can use taken branches to represent reached symbolic state. After we have identified all taken branches, the set of taken branches is then cached to represent the reached symbolic state. The comparison of two symbolic states is reduced to the comparison of two sets of taken branches.

It should be noted that not all taken branches on the corresponding subpath are needed in order to represent a symbolic state. For example, some branches lead to a constant assignment to a variable, which does not influence the symbolic state. We use a backward tracing method to identify all necessary taken branches. For a reached symbolic state at cycle \( k \), we transitively trace the executed assignment to register variables at cycle \( k \) backwards until primary inputs or constants are reached. That tracing process exactly follows the corresponding subpath. The taken branches, on which any traced assignment is control dependent, are retained to represent the reached symbolic state. In general, this tracing process does not have to go back to the initial cycle because (1) a constant is assigned to a variable on the subpath, (2) the primary inputs in the intermediate cycle may be used to define a variable on the subpath.

The use of branches to represent symbolic state is a conservative method. It means that two symbolic states may be identical although their sets of taken branches in concrete paths are different. The comparison between two symbolic states requires an SMT solver, as mentioned earlier. In this sense, our method is not able to completely prune the paths starting from previously explored symbolic states. Two symbolic states represented by two different sets of branches can be the same states, a case that our method is not able to detect.
3.7.4 Encoding of Explored Symbolic State

In order to efficiently store and compare sets of taken branches, we organize the taken branches of a reached symbolic state as bitmaps. Each cycle has a branch bitmap. If the branch is used to represent the symbolic state, its corresponding bit in the bitmap is set. A reached symbolic state in cycle $t$ can be represented by several bitmaps of continuous cycles ending at cycle $t$. When we check a reached symbolic state against previously explored symbolic state, the branch bitmaps of the reached symbolic state are also identified through use of the backward tracing method. The branch bitmaps of the reached symbolic state are then compared with the bitmaps of cached explored symbolic state.

Figure 3.6: Bitmap encoding of symbolic state. S7 is cached as explored symbolic state when path p1 is being explored. S7 is reached again in path p2.

Figure 3.7: The algorithm flow of STAR with explored symbolic state caching. The blocks in blue represent the steps of our explored symbolic state caching method.
When comparing a reached symbolic state at cycle $r$ with an explored symbolic state at cycle $k$, we require $r \geq k$. Without that requirement, the subpath starting from the reached symbolic state at cycle $r$ would be longer than any subpath starting from the same explored symbolic state at cycle $k$. New symbolic states may be reached by following a subpath starting at cycle $r$. Therefore, we should not prune the subpaths starting from cycle $r$.

In the example shown in Figure 3.6, the explored symbolic state is $S7$ of cycle $t$ in sequential path $p1$. The branch bitmaps are also generated for cycle $t - 1$ and cycle $t - 2$. When we explore the sequential path $p2$, we can check all the reached symbolic states in each cycle and compare them with cached and explored symbolic states. The symbolic state $S7$ was previously cached, and the corresponding cycle $t + 1$ is larger than the cached symbolic state’s cycle number $t$. We can prune all the subpaths starting from $S7$ in sequential path $p2$. All guards after cycle $t$ are directly popped out from the constraint stack, and the guards in cycle $t$ are chosen for mutation.

3.8 Experimental Evaluation of Method II

According to the flow shown in Figure 3.7, we have implemented the explored symbolic state caching in STAR, and also all optimization strategies, with C++, which interacts with VCS simulator through the direct programming interface(DPI) and with Yices [57] constraint solver through its C Library Interface. All the following experiments are performed on a machine with four Intel i5 2.67GHz processor cores with 16GB of memory running Linux. We present a set of experimental results for the RTL models from ITC99 and OpenRISC1200 [1]. The OR1200-0/1/2 designs are instruction cache controller, data cache controller and Wishbone bus interface.

3.8.1 Comparison with Original STAR

The second experiment, for which the results are shown in Table 3.3, compared the enhanced STAR with the original STAR algorithm. We set the maximum runtime to

\footnote{We refer to the work in this chapter that augments STAR with explored symbolic state caching, bitmap encoding of taken branches and the two optimizations as the \textbf{enhanced STAR in the experimental results}.}
Table 3.3: Comparison between the enhanced STAR introduced in this paper and the original STAR. All runtimes are in seconds. The length of each generated pattern is equal to the unrolled depth. The runtime limit is set as one hour. The original STAR is not scalable for most designs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Cycles</th>
<th>Original STAR</th>
<th>Enhanced STAR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Runtime</td>
<td>Num. of Patterns</td>
</tr>
<tr>
<td>b01</td>
<td>10</td>
<td>0.76s</td>
<td>512</td>
</tr>
<tr>
<td>b06</td>
<td>10</td>
<td>175.77s</td>
<td>131072</td>
</tr>
<tr>
<td>b10</td>
<td>10</td>
<td>237.01s</td>
<td>70593</td>
</tr>
<tr>
<td>b11</td>
<td>50</td>
<td>&gt;3600s</td>
<td>-</td>
</tr>
<tr>
<td>b11</td>
<td>100</td>
<td>&gt;3600s</td>
<td>-</td>
</tr>
<tr>
<td>b14</td>
<td>10</td>
<td>&gt;3600s</td>
<td>-</td>
</tr>
<tr>
<td>or1200-0</td>
<td>50</td>
<td>&gt;3600s</td>
<td>-</td>
</tr>
<tr>
<td>or1200-0</td>
<td>100</td>
<td>&gt;3600s</td>
<td>-</td>
</tr>
<tr>
<td>or1200-1</td>
<td>100</td>
<td>&gt;3600s</td>
<td>-</td>
</tr>
<tr>
<td>or1200-2</td>
<td>10</td>
<td>&gt;3600s</td>
<td>-</td>
</tr>
</tbody>
</table>

one hour. It can be observed that STAR suffers from path explosion for most of the designs. That is shown by long runtime over 1 hour. The runtime will increase exponentially as the unrolled depth increases in STAR. Therefore, the unrolled depth in this experiment was very small in original STAR. Thus, the original STAR is unscalable for big, practical designs. For the same circuit, however, the runtime for enhanced STAR was very small. The runtime of enhanced STAR does not increase exponentially with the unrolling cycles. The average number of branches repeated by the generated test set is used as a measurement to approximately reflect repetitive covering of the same design function. It can be observed that the number of repeated branches is very high in the original STAR, whereas in the enhanced STAR, the average number of repetitively covered branches is less than 6% of that in original STAR.

3.8.2 Optimization Effectiveness of Enhanced STAR

The optimization detail column in Table 3.4 shows the optimization effectiveness of UD chain slicing and local conflict resolution. The explored symbolic state caching’s effect is also shown in the “caching” column. UD chain slicing reduces the number of constraints sent to the SMT solver, since unused variables’ definitions will be excluded from current constraints. Local conflict resolution reduces the number of calls to the SMT solver, since it takes advantage of the static analysis
Table 3.4: The “UD chain slicing” column represents the percentage of reduced constraint numbers. The “local conflict resolution” column represents the number of conflicts detected during the mutation of constraints. The “caching” column represents the number of detected explored symbolic states.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Cycles</th>
<th>UD chain slicing</th>
<th>Local conflict resolution</th>
<th>Caching</th>
</tr>
</thead>
<tbody>
<tr>
<td>b01</td>
<td>10</td>
<td>56.7%</td>
<td>31</td>
<td>24</td>
</tr>
<tr>
<td>b10</td>
<td>15</td>
<td>43.4%</td>
<td>224</td>
<td>515</td>
</tr>
<tr>
<td>b11</td>
<td>100</td>
<td>5.24%</td>
<td>1245</td>
<td>720</td>
</tr>
<tr>
<td>or1200-0</td>
<td>100</td>
<td>0.2%</td>
<td>759</td>
<td>370</td>
</tr>
<tr>
<td>or1200-1</td>
<td>100</td>
<td>0.22%</td>
<td>280</td>
<td>86</td>
</tr>
<tr>
<td>or1200-2</td>
<td>10</td>
<td>23.4%</td>
<td>178</td>
<td>11850</td>
</tr>
</tbody>
</table>

Table 3.5: The coverage, running time, number of patterns and repeated branches reported by the enhanced STAR. The generated tests by our enhanced STAR have high structural coverage as well as functional coverage. The enhanced STAR is also compared with constraint-based random test generation method. The tests generated by the enhanced STAR have much higher coverage than the tests generated by constraint based random test generation method.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>STAR with explored symbolic state caching</th>
<th>(Constraint-based) random test generation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycles</td>
<td>Bran_Cov</td>
</tr>
<tr>
<td>b01</td>
<td>10</td>
<td>94.44%</td>
</tr>
<tr>
<td>b06</td>
<td>10</td>
<td>94.12%</td>
</tr>
<tr>
<td>b10</td>
<td>10</td>
<td>87.10%</td>
</tr>
<tr>
<td>b11</td>
<td>50</td>
<td>96.77%</td>
</tr>
<tr>
<td>b11</td>
<td>100</td>
<td>91.30%</td>
</tr>
<tr>
<td>b11</td>
<td>150</td>
<td>91.30%</td>
</tr>
<tr>
<td>b14</td>
<td>10</td>
<td>83.50%</td>
</tr>
<tr>
<td>or1200-0</td>
<td>50</td>
<td>93.75%</td>
</tr>
<tr>
<td>or1200-0</td>
<td>100</td>
<td>93.75%</td>
</tr>
<tr>
<td>or1200-1</td>
<td>50</td>
<td>96.55%</td>
</tr>
<tr>
<td>or1200-1</td>
<td>100</td>
<td>96.55%</td>
</tr>
<tr>
<td>or1200-2</td>
<td>10</td>
<td>100%</td>
</tr>
</tbody>
</table>

to check the conflict in the constraints stack instead of sending them to the SMT solver. The UD chain slicing can reduce the number of constraints by 0.2%–56.7%. Shown in the “local conflict resolution” column, this optimization is able to find local constraint conflicts and successfully avoid unnecessary calls to the SMT solver. For explored symbolic state caching, it can be observed that there are 24–11850 symbolic states on different designs identified as having been explored before.
3.8.3 Coverage Evaluation

Structural Coverage Evaluation

The first experiment in Table 3.5 shows the coverage rate for the generated test patterns by the enhanced STAR. It can be observed that the enhanced STAR can generate tests that achieve very high structural coverage as long as the unrolled depth is enough.

The unrolled depth is an important parameter for improving coverage in STAR. It can be determined by design engineers or by coverage feedback. If the coverage is not high, it means that the uncovered parts are not reachable within the unrolled design. We can increase the unrolled depth. The $b_{10}$ and $b_{11}$ circuits demonstrate that relationship between coverage and unrolled depth. When the unrolled depth increases from 10 to 30, all feasible branches are fully covered. However, for or1200-2, 10 cycles is enough to cover all branches. The running time demonstrates the applicability of the enhanced STAR for practical circuits. There is no memory bottlenecks even if we store the explored symbolic state in the form of bitmaps.

It is worthy of mentioning that infeasible paths can be identified during path enumeration. This is highly valuable to the verification engineer. In addition, the enhanced STAR can also be used to check properties on each path.

Functional Coverage Analysis

We also demonstrated that the input patterns generated by the enhanced STAR can provide high functional coverage although the entire analysis is on the RTL source code. The RTL code structures reflect the design specifications and STAR is able to follow these structures to generate meaningful patterns from the perspective of interesting functionality.

We used the assertion generation tool GoldMine [29] to generate assertions for several designs. The generated tests from the enhanced STAR were applied to the design to evaluate the assertion coverage during concrete execution. The assertions are normally used to check whether the design has correctly implemented the specification. The triggered assertion coverage rate reflects the quality of simulation patterns. The assertion coverage report on the tests generated by the enhanced STAR is shown in the “assertion coverage” column of Table 3.5. From the results, we can conclude that the enhanced STAR is able to generate tests that comprehen-
sively capture the design specification.

In comparison with random generation method, the enhanced STAR has the advantage of being able to generate all meaningful input to cover all possible design behaviors, instead of exhausting all possible values of inputs. The generate process relies on the hints from the RTL structure. Unlike widely used constraint-random generation method, our enhanced STAR doesn’t have to build the constraints manually.

3.8.4 Comparison with Constraint-based Random Test Generation Method

In this experiment, we compare the enhanced STAR with constraint-based random test generation method. We use both methods to generate tests and compare the corresponding structural and functional coverage values. Constraint-based random test generation method is widely used in simulation based verification. It requires the user to express the legal/valid input requirements in the form of constraint in verification testbench. The generator employs a constraint satisfaction technique to solve the complex constraint and produce input stimuli satisfying the input requirements. For OpenRISC design, we adopt the constraint-based random test generation testbench from the opencore website [1]. For ITC99 benchmark, we generate tests by randomizing the input variables in RTL designs since we did not find published testbench for these designs. We generate 1000 random patterns with different number of cycles for ITC99 benchmark.

We evaluate the structural coverage as well as functional coverage of both methods. From the experimental results shown in Table 3.5, it can be observed that the tests generated by the enhanced STAR always has higher coverage than the tests generated by other random method. Especially, for the design b14 with a lot of control branches, it is very difficult for random methods to generate high coverage tests. Also for constraint based random tests, we found that it always trigger a small number of paths repeatedly while it is not able to generate tests to cover the whole design. However, the enhanced STAR is able to generate high coverage tests.
3.9 Summary

We have presented a scalable, efficient input vector generation strategy that provides very high structural and functional coverage. The main novelty of this technique is in the hybrid analysis between concrete simulation data and static analysis of the RTL code and the heuristical branch guided path exploration. We believe that this technique is highly powerful and can be applied to large scale contemporary designs.
CHAPTER 4

TOWARD COVERAGE CLOSURE: USING GOLDMINE ASSERTIONS FOR GENERATING DESIGN VALIDATION STIMULUS

4.1 Introduction

Simulation-based verification relies on simulating the directed tests or constraint-random generated tests on the design and then checks the response against specification\(^1\). Although directed tests capture much of the desired system behavior, they do not suffice in checking for unintentional erroneous behavior. A phase of random input vector generation is employed with an intention to capture infrequent or unexpected design behavior. Due to the practical infeasibility of exhaustive simulation, the termination point of random simulation is very nebulous. Contemporary industries often use a numeric value like a few million simulation cycles before concluding the random simulation phase. Evidently, such a methodology is unsystematic and inconclusive.

In order to evaluate the comprehensiveness of the simulated tests and the degree to which the design has been simulated, coverage metric is utilized in simulation to provide a quantitatively measurement. Multiple types of coverage metric are available in state of art: code coverage, structural coverage and functional coverage \cite{4}. Despite various coverage metrics, there is no assurance that there are no gaping holes in the design behavior. Coverage closure, or the process of determining the completeness of functional coverage of input vectors, is therefore one of the most daunting challenges of the present day validation environment.

In this chapter, we propose a methodology for attaining coverage closure of design validation. The methodology is based on GoldMine, an automatic assertion generation tool that was introduced in \cite{29}. GoldMine uses data mining and static analysis to generate assertions. A Register Transfer Level (RTL) design is simulated and the simulation traces are passed as data to GoldMine. GoldMine uses decision

\(^1\)The word \textit{test} in this thesis refers to the simulation of functional stimuli on the design instead of manufacturing level testing.
tree based supervised learning algorithms to mine rules from the simulated test data. Each learned rule is considered as a candidate assertion. In order to determine if a candidate assertion is true for all inputs or not, the candidate assertions is passed with the design to a formal verification engine. If the formal verification passes a candidate assertion, it is a system invariant. If not, it generates a counterexample. We showed generation of complex and useful propositional (combinational) as well as sequential (temporal) assertions by GoldMine.

We incorporate feedback from the counterexamples generated by GoldMine to refine the simulation data that was used to generate assertions. The test data refined by counterexamples is now used to run another pass of GoldMine. The counterexamples from assertions that fail formal verification are again fed back into the input test suite. This iterative refinement continues until a pass of GoldMine where all the assertions pass the formal check. The test suite that remains at that point, along with the passing assertions, is the output of our method. We introduce a variation on the original decision tree data structure that is built incrementally with every iteration. An incremental decision tree per output adds information from a counterexample for every failed assertion on its leaf nodes.

Let us now look at how this counterexample guided automatic assertion/test generation process attains coverage closure. Firstly, in GoldMine, we stipulate that only 100 per cent confidence candidate assertions need to be considered for formal verification. Even a single contradicting example in the simulation data is enough to discard a candidate assertion. This ensures that a failing assertion that produced a counterexample is never reproduced by GoldMine in successive iterations. Since every counterexample provides a trace through the system and the addition of new variables, the corresponding input vector tests for as yet uncovered behavior. Every iteration, therefore, increases coverage of the test suite. This results in a monotonic decrease in the design space uncovered by the tests with successive iterations. In stark contrast to random or directed testing, where arbitrary long phases of coverage stagnation can occur, our method always makes forward progress with respect to test coverage.

Secondly, the limiting condition for this algorithm to converge is when there are no failing assertions. At this point, for every decision tree corresponding to a design output, all the assertions in the leaf nodes are true. This provides a deterministic metric of progress for test development. Until all the assertions for a given output pass, the test suite can be improved upon.

Thirdly, our method also provides an alternative notion of coverage- one that is
output-directed. If all the leaves of a decision tree have true assertions, it implies that the (incremental) decision tree now captures the complete functionality of that output. The decision tree that was predicting design behavior by observing dynamic data has completely captured the output logic function at the convergence point. We provide a proof intuition for the correctness and convergence of our algorithm. The final decision tree partitions the input space of target output into several equivalent classes. Each equivalent class will be covered by at least one test pattern. Additionally, since the decision tree extracts information from dynamic, simulation data, it generates only the reachable state of an output in sequential design and it is not possible to reach illegal or unreachable states in our method. At the point of convergence, the input test patterns along with the GoldMine assertions represent the validation artifacts required for achieving coverage closure. We consider the validation task complete when the entire functionality of all outputs in the design have been captured, i.e. all the assertions for the outputs are true.

4.2 Counterexample-Based Incremental Decision Trees

The decision tree is a structure that captures the design model from the perspective of observable behavior. An assertion can be false due to two reasons- either some behavior has not been observed by the decision tree due to insufficient data, or some inference has been made erroneously due to selecting a correlated, but not causal splitting variable. A counterexample trace exposes both these situations by introducing scenarios that involve at least one new variable. If this new scenario is now included in the input pattern data observed by the decision tree, firstly it prevents the generation of the same spurious assertion. Secondly, it guides the decision tree to navigate regions of input space that have not been considered/observed so far. A beneficial side effect of this process is the increase in coverage of the input simulation data steadily with every iteration.

In order to disprove an assertion, the new data instance consists of all antecedent variables of the assertion and some new additional variables. The antecedent variables’ values are also identical to that in the false assertion and the implied variable’s value is different from that in the false assertion. This characteristic of a counterexample enables a natural way to add it as new data instance to incrementally build a decision tree instead of rebuilding a decision tree from scratch every iteration.
Figure 4.1: Flow of counterexample-based incremental decision tree algorithm for generating validation stimulus in GoldMine.

```
1. Incr_Decimal_Tree_Building(TreeNode Node)
2. begin
3. if(Error(Node)==0) begin
4. if(Formal_verfn(Node.assertion)==true)
5. return;
6. else begin
7. Ctx_simulation();
8. Recompute_error(Node);
9. end
10. end
11. Node.left=New_node();
12. Node.right=New_node();
13. Select_splitting_variable();
14. Incr_Decimal_Tree_Building(Node.left);
15. Incr_Decimal_Tree_Building(Node.right);
16. end
```

Figure 4.2: Incremental Decision Tree Algorithm. The dotted lines represent parts that are different from GoldMine’s decision tree building approach.

In order to keep track of the improvisation of the decision tree for a given output, we devised an incremental version of the decision tree. The iterative algorithm using GoldMine (depicted in Figure 4.1) incrementally builds a decision tree for an output until it reaches the goal of generating only true assertions (no counterexamples). The full set of correct assertions, plus the new test patterns created from counterexamples during iterations, comprise the tangible outputs of the algorithm.

In the recursive incremental decision tree algorithm described in Figure 4.2, the parts different from GoldMine (lines 4, 7, 8) are outlined. Figure 4.3 shows the a regular decision tree and an incremental version of it.

A decision tree corresponds to a design output. The formal verification in line 4 is employed to check the correctness of assertion whenever a leaf node is reached during the incremental building of decision tree. If a candidate assertion is true
on design, the algorithm returns as in the regular decision tree. In the example, assertions $A_1$ and $A_2$ generated from original simulation traces are true on the design. If the checked assertion is false/spurious, a counterexample is reported by formal verification. A counterexample, $a = 0, b = 1, c = 0$ and $z = 1$, is generated to contradict the assertion $A_0$ on the decision tree on the left.

The Ctx_simulation() function simulates the input pattern created by the counterexample. This lends concrete values to all the splitting variables in previous iterations of the decision tree in the new simulation run.

Since the counterexample follows the same path as the failed assertion, the decision tree continues splitting when it reaches the leaf node corresponding to that false assertion. All other paths of the decision tree are kept unchanged. Due to the new data instance, the mean and error values for each node need to be recomputed using the Recompute_error() function. The error value of the leaf node will no longer be equal to zero. In the example, the incremental decision tree continues to split on the leaf node corresponding to false assertion $A_0$ in the regular decision tree. It can also be observed that the mean and error value are recomputed in this iteration on the path from the root to the leaf. The algorithm exits when all the assertions at the leaf nodes of an incremental tree are true.
4.2.1 Stimulus Generation for Sequential Behavior

During the building of a decision tree, the design should be unrolled until the mining window length, as defined in Section 2.2.1. The simulation trace used for assertion mining may have internal register state visible. It may be desirable to have assertions form a single-cycle flat picture of the design, where assertions on the outputs are functions of internal state values and primary inputs. Assertions can also be formed for the internal state variables themselves, as functions of other state registers and inputs. Such a view of the design gives a “next cycle” model, where the assertions describe internal registers and primary outputs in a similar manner. On the other hand, it may be desirable to have temporal assertions on the design that capture only input-output behavior over some number of cycles.

We can generate assertions of both types with this algorithm, based on the mining window length and visible state provided. Although the assertion spans sequential behavior over a given length, the generated counterexample may be longer than the mining window length. This may be to expose sequential behavior where an intermediate state variable can be driven to a specific value over several cycles starting from the primary input. In this case, the incremental decision tree algorithm considers only the state variables until the farthest back temporal stage, i.e. unrolled until the mining window length. The concrete values of these variables can be acquired through simulation of the counterexample by the data generator. The result is a temporal assertion that spans the mining window length, bolstered by single-cycle assertions using internal state registers to describe the behavior. We discuss an example of sequential logic coverage in Section 4.5.

The length of mining window will limit the generated form of assertions, but it has no impact on the full coverage of generated assertions or tests on the whole design. By using mining window, only primary inputs and state variables until the farthest back temporal stage in mining window are considered as decision variables for a target output. However, these state variables are also individually set as target outputs for assertion generation. As a result, the relations between variables outside mining window and target output are connected by the assertions on these intermediate state variables. In other words, we can use all assertions generated for all primary outputs and state variables within mining window to form longer assertions. Similarly, we will also generated tests for these state variables. Therefore, from the perspective of all primary outputs and state variables, full coverage of generated tests and assertions on the whole design can be guaranteed.
4.2.2 Final Decision Tree

Our counterexample based incremental decision tree building algorithm is a process of approximation and refinement of an output function. If the complete functionality of an output was available to the decision tree in the form of simulation data, it would completely represent the output function. Such a truth table (or state transition relation for sequential designs) would result in a complete decision tree. However, such an exhaustive enumeration of input patterns is not feasible to obtain as test data. Therefore, the decision tree tries to approximately predict the logic function of an output with available data. Faulty predictions are exposed and used for corrective purposes through counterexamples. This makes future predictions more accurate. The point where all the predictions are accurate is where all the assertions of the decision tree are true. At this point of convergence, this final decision tree represents the complete functionality of an output in the design. It is important to note that final decision trees include only the legal, reachable states of the design. The reachability analysis of decision tree is presented in Section 4.3. The input patterns required to generate such a final decision tree are actually sufficient for completely covering the functionality of that output. This coverage notion is formally presented in Section 4.4.

4.3 Algorithm Completeness and Convergence Analysis

In this section, we prove that our counterexample based test generation algorithm converges and at the point of convergence for any output, the corresponding decision tree for that output represents the complete functionality of that output.

We present some definitions that are required for proving the mentioned claims. Let us consider an RTL design whose state transition graph (Kripke structure [94]) model is depicted by $M$. We will use $M$ synonymously for the design as well its model. Let there be $N$ inputs in $M$. An input pattern is a unique assignment of values to inputs of $M$. Input patterns can be combinational (single cycle) or sequential (across multiple cycles). An input pattern set is a set of all such input patterns in use for a design validation effort.

The input pattern set for $M$ forms the data for the decision tree algorithm. We define decision trees as used in our context.
Definition 1 A decision tree $D_z$ for an output $z$ is a binary tree where each node corresponds to a unique splitting variable that is statistically correlated to $z$. A path for a decision tree is a sequence of nodes from the root node to a leaf node.

In general, decision trees need not be binary trees, but since our variables are in the Boolean domain, there are only two possible values of each (one-bit) variable. A decision tree is a data structure used in predictive modeling to map observations about a variable of interest to inferences about the variable’s target value. In our case, every output of $M$ is a variable of interest. Every output has a corresponding decision tree that makes inferences about the output’s target value (true and false). These inferences are made at the leaves of the decision tree, where the branches leading from the root to the leaf represent conjunctions of splitting (correlated) variables. These inferences are also considered likely or candidate assertions for the concerned output.

Definition 2 A candidate assertion $A_C$ of $D_z$ is a proposition consisting of an antecedent and a consequent. The antecedent is a Boolean conjunction of propositions $(\text{variable, value})$ pairs along a path in $D_z$. The consequent is also a proposition: $(z, \text{value})$ pair where the value of output $z$ is the mean value on the path’s leaf node.

In the next phase of GoldMine, model checking $[5]^2$ is used to compute the truth or falsehood of a candidate assertion. In case a candidate assertion is false, a counterexample or simulation trace through the design is generated, that exemplifies the violation of the assertion.

Definition 3 A true assertion $A_T$ is a candidate assertion such that $M \models G(A_T)$. In other words, true assertion $A_T$ globally holds on model $M$.

Definition 4 The support of a Boolean conjunction $y$, which is denoted as $\text{support}(y)$, is the set of variables in $y$.

Definition 5 If $M \not\models G(A_C)$, the conjunction of variable value pairs in the counterexample is represented by $\chi_{A_C}$ such that $\text{support}(\chi_{A_C}) \supset \text{support}(A_C)$.

$^2$We categorize the formal verification algorithms in SMV and Cadence IFV under the umbrella of model checking for this discussion.
Since the counterexample represents a valid simulation trace through the design that is not yet a part of the current input pattern set, it is added to the input pattern set. An incremental version of the decision tree is used in order to keep track of the coverage. The incremental decision tree maintains the ordering of variables as the decision tree from a previous iteration for all the variables until the leaf nodes. If the counterexample in the current iteration coincides with a path in the incremental decision tree, the variable(s) added by the counterexample will now be used as the splitting variable(s) at the leaf nodes of the incremental decision tree.

**Definition 6** An incremental decision tree $I^z$ for an output $z$ and a previous decision tree $D^z$, is a decision tree such that the variable ordering of all variables in $D^z$ is preserved until a leaf node. Every variable $v$ in $\text{support}(\chi_{AC}) - \text{support}(AC)$ becomes a splitting variable at the leaf node of $I^z$ along the path of $AC$.

**Definition 7** The final decision tree $F^z$ is an incremental decision tree such that for all assertions $AC$ of $F^z$, $M \models G(AC)$.

**Definition 8** The logic cone of an output $z$ in $M$ is the set of variables that affect $z$.

The logic cone is deciphered by computing the transitive closure of all variables pertaining to an output. In GoldMine, we do a logic cone analysis for every output. The decision tree for an output is therefore restricted to the variables in its logic cone, or the relevant variables with respect to that output.

**Theorem 1** It takes finite iterations to reach $F^z$ for any given $I^z$.

*Proof:* Let us run the incremental algorithm for $k$ iterations, then the maximum number of new nodes added to $I^z$ is $2^k$. The maximum total number of nodes in $I^z$ after $k$ iterations is $2^{k+1}-1$. Let $n \subseteq N$ be the number of variables in the logic cone of $z$. The maximum size for $D^z$ by construction and by definition of binary trees is $2^{n+1}-1$. Therefore, $2^{k+1}-1 \leq 2^{n+1}-1$. This bounds the size of the incremental decision tree.

It may be noted that since we are restricting the decision tree for an output to focus only on the relevant variables, the maximum size of the decision tree is not exponential in the size of the entire set of inputs $N$, but in $n$. In practice, we observe that $n \ll N$.

**Theorem 2** The final decision tree $F^z$ corresponds to the entire functionality of $z$. 

59
**Proof:** Assuming a final decision tree $F^z$ does not correspond to the entire functionality of $z$, then there is at least one input pattern to reach a state of $z$ that does not correspond to a path in $F^z$, so at least one $A_C$ of $F^z$ should be such that $M \not\models G(A_C)$. But this is false by definition of $F^z$. Therefore, the assumption is contradicted.

The above theorems make a powerful statement about the coverage of our method. When all the assertions are true, the complete functionality of an output is captured. These theorems are applicable to both combinational and sequential behaviors. For sequential behavior, we can unroll the circuit for mining window length and each variable in different cycle is annotated with cycle index. Therefore, it can also be viewed as a combinational logic. The only difference is that the number of input variables in logic cone of target output will linearly increase with the mining window length.

Specially for sequential design, the final decision tree is able to include only reachable states and all generated assertions are non-vacuous. Reachable states are the union set of all possible states that can be reached from initial states. In general, reachable states are a subset of whole state space due to the constraints on state variables. We will formally prove these features of final decision tree in Section 4.8.

### 4.4 Coverage Analysis

In the simplest terms, what we want from a coverage effort is exposure of the entire legal, reachable design behavioral space to examination so that this space can be validated against a statement of desired behavior. We posit that our algorithm using GoldMine and iterative refinement of the decision tree achieves exactly that property: when the final decision tree for an output has been constructed, the input patterns or assertions generated by decision tree are artifacts that represent the complete functionality of that output. Our notion of coverage, then is output directed, as opposed to traditional assertion based coverage or code structure based coverage [95]. For each such target output, we consider the truth table description of the output behavior where each input combination corresponds to a value for the output. The coverage referred to per output, then, is the input space coverage, *i.e.* the number of truth table entries covered. The entire input space of a target output
is divided into several classes. We assert that the generated GoldMine tests are able to cover each class at least once. With respect to this notion of coverage, we can achieve functional coverage closure for every output in the design.

4.4.1 Coverage Definition

We provide some definitions that elucidate our notion of coverage for tests. The coverage notion of our generated tests is based on all entries in the truth table of a target output. We define a test to have input space coverage if it corresponds to an entry in the truth table of the target output. The complete functionality of a target output is exactly represented by the truth table.

Now let us take a look at the mechanics of how the GoldMine test generation process is able to partition the input space of a target output. The decision tree in every iteration partitions the input data set (tests) until it can find a “fit”, i.e. it reaches the leaf node. The partitions are created on the basis of data values per variable. Since the data values are Boolean for single bit variables, the partitioning criteria be thought of as propositions (true/false values of a variable). Every branch in the decision tree partitions the input data on the basis of a proposition being true or false. Every successive level of the tree partitions the input space further, until it reaches a leaf node. In terms of propositions, the “path constraint” in a tree leading from the root to a leaf is a conjunction of the corresponding propositions from root to leaf node. Each path from the root to the leaf node, then, represents a partition of the input test set. When this decision tree is grown incrementally, the final decision tree will partition the final set of tests. Then, every path from root to leaf divides the tests into an equivalence class, such that the path constraint is the same for all the tests in the class.

The decision tree is not deterministic in its decisions. It makes optimistic decisions in the dataset, and predicts that a certain splitting variable is related to the target output. If this optimistic prediction is true, then it uncovers a relationship among input variables and the target output. This relationship usually has fewer variables than the tests from which it was derived. Hence the tests created by feeding back the true assertion relationships into the test suite are more compact than the tests used initially. This is the reason why the test suite at the end of the final decision tree generation is not just complete, but much more compact than the entire truth table representation for the target output. For example, if in the function $z = a \land b$
the initial test set is \(<a = 0, b = 0>, <a = 1, b = 0>\) and \(<a = 1, b = 1>\). The decision tree will make an optimistic prediction that \(a = 0 \Rightarrow z = 0\). This true assertion leads to the test \(<a = 0, b = X>\), that now covers the table entries: \(<a = 0, b = 0>\) and \(<a = 0, b = 1>\) for output \(z\).

Due to the optimistic prediction, the path constraint between the root and leaf node can correspond to multiple truth table entries. In the above example, the constraint \(a = 0\) corresponds to two truth table entries \(<a = 0, b = 0>\) and \(<a = 0, b = 1>\). We established above that the input test set is partitioned by a path constraint. Therefore, the path constraint of a leaf node represents a set of tests as well as a set of truth table entries. We can then say that in a final decision tree for a target output, the tests represented by a leaf node completely cover the truth table entries that correspond to the path constraint of the leaf node. This is our notion of coverage.

The final decision tree partitions the input test data into several equivalence classes. Each leaf node corresponds to a set of truth table entries. This partition is exclusive or the set of truth table entries at one leaf node does not intersect with any other leaf node; and complete, or the union of all sets of entries on all leaf nodes of the tree consists of the full truth table. We prove these two properties.

Consider a design \(M\) with input alphabet \(\Sigma\) and outputs \(O\). Let a target output be \(z \subset O\), and \(I \subset \Sigma\) where \(I\) is the set of variables in the logic cone of \(z\). Let the input test set for the final decision tree \(D_z\) for \(z\) be \(\tau\), which includes initial tests and the generated counterexamples during incremental process. Let \(N\) be the set of leaf nodes and \(E\) the set of edges in \(D_z\).

In a decision tree, each edge is associated with a unique true or false proposition with respect to a variable in \(I\). Each proposition is a \((\text{variable}, \text{value})\) pair. Each node of the decision tree can be reached by following a path (sequence of edges) starting at the root.

**Definition 9** The conjunction of propositions on every edge from the root of a decision tree to a target node is the path constraint of that node. We denote the path constraint of a node \(n\) as \(P_n\).

If \(n\) is a leaf node, \(P_n\) is the same as the antecedent of a GoldMine assertion and does not include the consequent (Definition 2).

The output function \(f\) such that \(z = f(I)\) can be represented as a truth table. Each truth table entry is a unique set of values for variables in \(I\). Let \(\Theta\) be the set of all truth table entries for \(f\). The set of variable assignments in a truth table entry
can be examined for consistency (or satisfiability) with respect to a path constraint in the decision tree.

**Definition 10** A truth table entry $\theta$ is said to be consistent with a path constraint $P_k$ if for all common variables $v$, the values in $\theta$ and $P$ are equal. Then, for every variable in $\theta$ that belongs to $I \setminus v$, the value is $X$.

Let $\theta_n$ be the set of all such truth table entries that are consistent with path constraint $P_n$ for a leaf node $n$ of $D_z$. We can say the truth table entries $\theta_n$ correspond to node $n$.

**Lemma 1:** The path constraint of a leaf node is unique.

Let us assume the path constraint is non-unique. Consider two leaf nodes $i$ and $j$, such that $i \neq j$. Then by the definition of path constraint, the conjunction of propositions from root to node $i$ would coincide with the conjunction of propositions from root to node $j$. Given that every edge has a unique proposition, this cannot happen unless $i$ and $j$ are equal. So there is at least one edge that diverges to reach the other leaf node.

We want to show that the truth table entries corresponding to two different nodes are not coincident. Or, that a truth table entry can correspond to exactly one leaf node.

**Theorem 3** $\forall$ leaf nodes $i, j \in N$ and $i \neq j$, $\theta_i \cap \theta_j = \emptyset$.

**Proof intuition:** Let us assume that $\theta_i \cap \theta_j \neq \emptyset$. Then, there exists an entry $e$ such that $e \in \theta_i$ and $e \in \theta_j$ for different $i, j$. This means that $e$ is consistent with $P_i$ as well as $P_j$ (from Definition 10). From Lemma 1, this means that $i$ and $j$ are coincident. Hence, there is a contradiction.

We now want to show that the union of the truth table entries corresponding to all the leaf nodes in the final decision tree is exhaustive.

**Theorem 4** Let $\theta_i$ be a subset of the entries in $\Theta$. For a final decision tree $D_z$ for output $z$,

$$\Theta = \bigcup_{i \in N} \theta_i.$$

**Proof intuition:** By construction, $\bigcup_{i \in N} \theta_i \subseteq \Theta$. To show equality, we need to show that $\bigcup_{i \in N} \theta_i \supseteq \Theta$, i.e. given any entry $e$ in $\Theta$, $\exists i \in N$ and $e \in \theta_i$. We need to show that each truth table entry corresponds to a leaf node in the decision tree. For any entry $e$, we can find if it is consistent with a path constraint $P_k$ of a leaf node the
final decision tree. If it is consistent with $P_k$, then it corresponds to the leaf node $k$. If it is inconsistent with $P_k$, then there is at least one variable that has contradicting values between $e$ and $P_k$. From the construction of a decision tree, the contradicting value would be along an edge on a different path than that leading to $k$. This means that $e$ would be consistent with that path, and therefore correspond to its leaf node.

From the theorems above, we see that the final decision tree divides the truth table entries into several equivalence classes. Each equivalence class is a set of table entries and corresponds to one leaf node in the tree. Within each equivalence class, the table entries all have the same value on $z$.

Each test in $\tau$ used for decision tree building is a vector of assignments to all variables in $I$. If we simulate this test on the design, we can also generate a concrete value on $z$. It should be noted that some input variables may be $X$ in initial tests or counterexamples. The decision tree algorithm randomly assigns a concrete value on these don’t care variables.

**Definition 11** The test in $\tau$ is said to be consistent with path constraint $P_n$ if for all common variables $v$, the values in the test and $P_n$ are equal.

Let $\tau_n$ be the set of tests that are consistent with path constraint $P_n$ for a leaf node $n$ of $D_z$, we can also say the tests $\tau_n$ corresponds to node $n$.

**Definition 12** Given a leaf node $i$ in a final decision tree $D_z$ for an output $z$, the truth table entries $\theta_i$ corresponding to $i$, form an equivalence class such that all of them have the same value for $z$. The set of tests $\tau_i$ corresponding to node $i$ is said to completely cover this equivalence class.

In principle, we can virtually view truth table entries as the input space of target output, the final decision tree automatically divides the input space into classes. Each class can be thought as a design behavior of target output. There is at least one generated test to cover each class. The above theorems and definitions are also applicable to sequential logic and all truth table entries of $z$ would only include the reachable entries where assignments to state variables are reachable from initial states.

The input space (truth table) coverage of the tests on one leaf node is equal to the percentage of truth table entries on that node. We only need to know the number($m$) of variables included in the antecedent of corresponding assertion. We denote the total number of input variables in logic cone as $n$. Then the coverage will be calculated as $2^{(n-m)}/2^n=1/2^m$. In previous example, the coverage of $<a=0, b=X>$
is 50%. The total input space coverage of tests on all leaf nodes with true assertions is just the sum of coverage on every such leaf node due to the exclusive property.

4.4.2 Coverage Closure

GoldMine’s counterexample based approach for test generation ensures a monotonic decrease of the uncovered design space with each iterative refinement. In each iteration, the generated counterexample is able to cover a new design function which has not been covered before by previous patterns. The newly activated function can be in the form of conditional expression, branch or assignment statements in the RTL design. Moreover, the existence of a final decision tree as a goal provides a deterministic metric of progress through the refinement process. This is a significant improvement over random testing, whose coverage graph can be arbitrarily shaped, often resulting in plateaus where no progress is being made. In fact, due to the frequent lack of feedback in the random test generation process, it is difficult to acquire a satisfactory functional coverage picture in this process.

Figure 4.4: The coverage of input patterns in the functional design space for an output.

A pictorial example of this process is shown in Figure 4.4. The state space for a single output can be visualized as a discrete 2D plot, where the functional points covered by the starting input test patterns are marked. Each GoldMine assertion generated includes a set of variable-value pairs according to their statistical support in the patterns.
Every assertion is therefore shown to span a group of points in the output state space by rectangular boxes. This grouping by assertions into “regions” in the output space is similar to a Karnaugh map notation, but this includes sequential behavior as well. For the assertions that are true, the design region has been covered by the input test patterns in that iteration. For the ones that are false, there is always at least one additional design point that was uncovered by the input test pattern. This design point is exposed by a violation of the assertion. Each counterexample (Ctx) acts a bridge between an uncovered design point in (a) and a covered design point as in (b). However, the covered design point in (b) forms a part of the region covered by an assertion, that generates a counterexample again. All previously true assertions do not perturb the coverage process and are retained in every phase. As a side effect, the original, general assertion is divided into multiple, more precise and subtle assertions.

We notice here that the GoldMine test generation strategy goes from uncovered regions in one iteration to covered regions in another, until it converges at all assertions passing as in (c). This is distinct from a traditional validation flow, where all the known regions are covered first, and an advancement is attempted toward uncovered regions.

4.5 Example: Two Port Arbiter

In this section, we will demonstrate GoldMine’s incremental counterexample refinement using a 2 port arbiter. This arbiter uses round robin logic with priority on port 0. In our example, we set the mining window length to two in GoldMine to generate temporal properties of the port 0 access signal, \( gnt_0 \). Each variable has cycle annotations \( t-1, t \) and \( t+1 \). In this example, the target output for which GoldMine generates assertions is \( gnt_0(t+1) \). Besides primary inputs, state variable until farthest back temporal stage(Section 4.2) \( gnt_0(t-1) \) is also included in the logic cone. When the tool outputs assertions, for any given signal \( sig, sig(t-1) \) is transformed to \( sig \), \( sig(t) \) is transformed to \( Xsig \) and \( sig(t+1) \) is transformed to \( XXsig \).

The simulation data shown in Figure 4.5 represents a directed test that a validation engineer might write. We will show how the A-Miner makes inferences about the design and is aided by the counterexample refinement to improve assertion and directed test quality.
always @ (posedge clk) if (rst) begin 
gnt0 <= 0; gnt1 <= 0; 
end else begin                                     
gnt0 <= (~gnt0 & req0) | (gnt0 & req0 & ~req1); gnt1 <= (gnt0 & req1) | (~gnt0 & ~req0 & req1); 
end

<table>
<thead>
<tr>
<th>req0 (t-1)</th>
<th>req0 (t)</th>
<th>req1 (t)</th>
<th>gnt0 (t)</th>
<th>gnt0 (t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.5: Arbiter: RTL and simulation trace.

In this example, we turn off the reset signal since we are interested only in normal behaviors under the condition that reset is not asserted. It may be noted that in cases where the reset behavior is interesting, we can generate assertions with the reset signal in the antecedent, by artificially applying the reset signal once every thousand cycles in the initial simulations. Our decision tree data structure starts with a root node which contains all examples and the examples are partitioned into likely behavior by the time they reach the leaf nodes. The initial structure of the decision tree is represented in Figure 4.6.

![Decision Tree Diagram]

Figure 4.6: Initial decision tree

A0: \(\neg req0 \Rightarrow XX \neg gnt0\)

A1: \(req0 \Rightarrow XX \neg gnt0\)

The two candidate assertions generated above are proven false by formal verification. A counterexample is produced for each failed assertion containing the series of states that will contradict this assertion. We simulate these counterexamples and add the results to our example set as show below. The decision tree continues to grow since the error is greater than 0 for each node. This means that the confidence is no longer 100% for A0 and A1. The A-Miner finds four more candidate assertions based on the new data.

A2: \(\neg req0 \land (\neg req0) \Rightarrow XX \neg gnt0\)
A3: \( \neg \text{req0} \land (X \text{req0}) \Rightarrow XX \neg \text{gnt0} \)

A4: \( \text{req0} \land (X \neg \text{req1}) \Rightarrow XX \neg \text{gnt0} \)

A5: \( \text{req0} \land (X \text{req1}) \Rightarrow XX \neg \text{gnt0} \)

After one iteration shown in Figure 4.7, A2 and A3 are verified to be true. However, A4 and A5 both fail formal verification and a counterexample is produced for each. We again simulate the counterexamples and add them to our data set. The refined tree is shown in Figure 4.8.

A6: \( \text{req0} \land (X \neg \text{req0}) \land (X \neg \text{req1}) \Rightarrow XX \neg \text{gnt0} \)

A7: \( \text{req0} \land (X \text{req0}) \land (X \neg \text{req1}) \Rightarrow XX \text{gnt0} \)

A8: \( \text{req0} \land (\neg \text{req1}) \land (X \text{req1}) \Rightarrow XX \neg \text{gnt0} \)

A9: \( \text{req0} \land \text{req1} \land (X \neg \text{req0}) \land (X \text{req1}) \Rightarrow XX \neg \text{gnt0} \)
A10: req0∧req1∧(Xreq0)∧(Xreq1)⇒XXgnt0

A6, A7, A8, and A9 are verified as true. However, A10 is shown to be false even though all primary inputs have been assigned. In this case, despite gnt0(t − 1) being in the logic cone of gnt0(t + 1), it is not included in the antecedent of the assertion. This is due to the fact that the error function on gnt0(t − 1) does not give a minimum value comparing with that on other variables. Consequently, the decision tree algorithm would not pick up gnt0(t − 1) for splitting. At this point, we allow the A-Miner to search the state variables in the farthest back temporal stage for a suitable split by incorporating the counterexample. In our example, we add the signal gnt0(t − 1) to the search. The A-Miner makes this split and produces the full tree below and A11 and A12 are newly generated true assertions, as shown in Figure 4.9.

![Figure 4.9: Third iteration: full tree](image)

A11: req0∧req1∧(Xreq0)∧(Xreq1)∧(¬gnt0)⇒XX(¬gnt0)

A12: req0∧req1∧(Xreq0)∧(Xreq1)∧gnt0⇒XXgnt0

After the assertions are generated by incremental counterexample refinement, the counterexamples can be added to the original directed test to improve coverage of the test. The series of inputs for each counterexample is simply added to the current input stimulation in the directed test. The improvement in expression coverage of each counterexample iteration is shown in Table 4.1.

In this example, the generated assertions or tests cover the temporal behavior within two cycles. However, these two cycles’ assertions can be used to form the assertions covering behaviors outside mining window. For example, by using antecedent conditions in A0 to replace the gnt0 in antecedent condition of A11, we can compose an assertion spanning to four cycles.
### Experimental Results

To evaluate the quality of our method, we implement the incremental decision tree building algorithm and generate validation stimulus and assertions for several design modules. These include some simple synthetic blocks we created to test various features, and some designs from the Rigel RTL design [96], the OpenRisc design [1] and the SpaceWire design [97]. In the rest of the experiment part, we will refer to our technique as counterexample based GoldMine tests. These designs are used for the following experiments:

1. Study the coverage increasing with the number of counterexample iterations

2. Limit studies of the counterexample method
   (a) Zero-pattern seed: Starting with no test patterns and iterate
   (b) Full-coverage seed: Starting with patterns that already provided 100% coverage using standard coverage metric

3. Bug finding: Injecting errors in RTL and using the generated assertions/stimulus as a regression suite to detect the injected errors

4. Comparison to standard coverage: Using current standard coverage metric to evaluate the test generated by our counterexample-based incremental method

We implemented the incremental decision tree algorithm using Java. Cadence IFV is employed as the formal verification engine and NC-verilog is used for simulation to generate a database for data mining. All experiments are run on an Intel Core 2 Quad Q6600 with 4GB of memory.

The runtime for this algorithm is proportional to the number of counterexample generated. The size of the design, number of initial samples, and maximum number of iterations all affect the number of counterexamples. In this experiment, most
test generation complete in 1-2 hours on our implementations. Memory usage is proportional to the number of examples and nodes. We can dynamically prune nodes at the end of the iteration. The used memory of all tests is below the 2GB.

4.6.1 Coverage Increase by Counterexample Iteration

The first experiment demonstrates the increase in coverage by assertions as the counterexample algorithm progresses, showing a monotonic increase in coverage. The experiment is performed on SpaceWire codec state machine circuit and Rigel write back stage design. The original test suite can be in the form of a directed test or a completely random input stimulus test. In this experiment, we simply use the initial random input patterns. In each iteration, any spurious assertions are refined using counterexamples until the A-Miner has generated a true assertion. The input space coverage and industrial standard coverage metric are used in this experiment. The input space coverage of each true assertion referring to corresponding output is calculated by considering the percentage of the truth table entries that is covered by that assertion. We have summarized these coverage results in Figure 4.10 and Figure 4.11.

In Figure 4.10, the input space coverage referring to each output was chosen to measure the validation process. Since each assertion compactly covers multiple concrete patterns of input space, we calculate the input space coverage referring to an output by accumulating the input space coverage of all generated assertions on that output. The results show a consistent increase in the input space covered by the assertions in each iteration.

In Figure 4.11, we tested the line, conditional, branch, toggle and FSM coverage of the counterexample generated tests. Redundant statements, unreachable states and other RTL characteristics often limit some kind of coverage to achieve 100%, but a steady increase in such coverage is an indicator of monotonic progress in the quality of the assertion/tests generated by our algorithm.

We also notice that the coverage increases quickly in the early iteration and slowly in the later iteration. However, in contrast to the tradition industrial flow, our method can guarantee coverage gain in each step until full coverage is reached. In the worst case, the maximum number of iterations required to reach full coverage is equal to the number of input variables in the logic cone of corresponding output since at least one variable is added to the original assertion as counterexample to
disprove the spurious assertion.

4.6.2 Zero Initial Patterns

The second experiment is a limit study showing that the counterexample algorithm works even when no original directed or random test suite exists. The lack of any patterns would begin the procedure with a simple assertion of the form “output always 0”. Figure 4.12 shows the increase in coverage for each design as the algorithm progresses. Even without initial test patterns, the counterexample method is able to create a test suite that achieves good coverage with few iterations. This indicates that this method may be a useful methodology to jump start a module design environment by creating many tests that can then be run on the testbench to check against the design specification.
Figure 4.12: Coverage increasing by iteration starting from zero pattern on SpaceWire-FSM design.

Table 4.2: Improvement on test suites that have high coverage according to standard metrics. The initial test suites have achieved high coverage on some standard metrics. Counterexample based GoldMine tests are still able to increase the coverage on other standard metrics. Line, Condition(Cond), Toggle, FSM, and Branch Coverage metrics are shown as standard coverage metrics.

<table>
<thead>
<tr>
<th>Design Name</th>
<th>Initial test patterns with high coverage</th>
<th>GoldMine tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcache ctrl</td>
<td>100% Line 98.87% Cond 98.77% Toggle 100% FSM 93.75% Branch 98.77%</td>
<td>Line 100% Cond 78.87% Toggle 98.77% FSM 100% Branch 93.75%</td>
</tr>
<tr>
<td>icache ctrl</td>
<td>100% Line 93.13% Cond 99.11% Toggle 81.25% FSM 96.55% Branch 96.55%</td>
<td>Line 100% Cond 93.13% Toggle 99.11% FSM 100% Branch 93.75%</td>
</tr>
<tr>
<td>write_back</td>
<td>98.18% Line 96.36% Cond - Toggle - FSM 96.30% Branch 100%</td>
<td>Line 100% Cond 49.76% Toggle 98.46% FSM 100% Branch -</td>
</tr>
</tbody>
</table>

As an extreme case in this experiment, the output is indeed always 0. There will be no test for this case using our algorithm since the first assertion has already captured the output function. From the perspective of design, all the logic designs for this output are redundant. We do not need to generate any tests to cover a redundant logic.

4.6.3 Improvement on Patterns that Have 100% Coverage with Standard Metrics

The third experiment explores how our counterexample based GoldMine test generation can improve the test sets that have high coverage according to standard coverage metrics. The metrics we consider are line coverage, conditional coverage, toggle coverage, FSM coverage and branch coverage [95]. It is well known that any one of these metrics are not sufficient in expressing the extent of coverage of input test patterns. To underscore the limiting case of this argument, we consider test patterns that have 100% coverage according to at least one of the standard coverage metrics. Our goal is to enhance these test suites by adding test patterns. This
is particularly useful when a block is declared as having been covered completely according to some tests, and detecting coverage holes is difficult. Table 4.2 shows that the data cache controller, instruction cache controller, and write back modules of the OpenRisc [1], whose tests report high line and branch coverage. Using our GoldMine test strategy, we are able to enhance the tests to increase the FSM coverage on data cache controller and instruction cache controller modules. For write back module, we enhance the line, condition, toggle and branch coverage. This experiment shows that (i) despite high coverage numbers using standard metrics, there is still scope for improvement in the coverage and (ii) this improvement can be detected and achieved using GoldMine tests.

4.6.4 Bug Detection by Generated Assertions/Stimuli

This experiment uses assertions to detect bugs in the design. We use a systematic mutation-based method to test the assertions’ ability to detect bugs. The RTL code is mutated and then all generated assertions are then formally check on the mutated design model. The failed assertions detect a corresponding bug on the mutated design. We inject four types of errors [98]: operator replacement, variable to constant replacement, constant replacement and relational operator replacement. For each output, we inject the errors into its logic cone and then formally check all generated assertions on mutated design. The experiment is conducted on the data/instruction cache control and write_back modules of OpenRisc [1]. For each injected error, there are always many assertions that can detect the error. Table 4.3 shows the number of injected errors and average percentage of generated assertions that can detect these injected errors. It can be observed that our generated assertions/stimuli effectively capture the potential bug in the design since every injected error will be detected by an average of more than 33% of the generated assertions. Although GoldMine targets at single output for assertion generation, the generated tests will be simulated on the whole design and may have impact on other outputs. The bug involving multiple outputs can thus be activated by our tests.

4.6.5 Comparison to Standard Coverage

In this experiment, we compare the counterexample generated test against directed tests using standard coverage metric. Final coverage values for three Rigel’s CPU
Table 4.3: Detecting of injected errors by assertions on OpenRisc module.

<table>
<thead>
<tr>
<th>Output Signal</th>
<th>No. of Assertions</th>
<th>No. of Injected Errors</th>
<th>Percentage of assertions detecting errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>biu_read</td>
<td>3260</td>
<td>4</td>
<td>60%</td>
</tr>
<tr>
<td>burst</td>
<td>1783</td>
<td>5</td>
<td>74.41%</td>
</tr>
<tr>
<td>dcram_we</td>
<td>388</td>
<td>10</td>
<td>67.78%</td>
</tr>
<tr>
<td>first_hit_ack</td>
<td>29</td>
<td>9</td>
<td>33%</td>
</tr>
<tr>
<td>hitmiss_eval</td>
<td>9</td>
<td>12</td>
<td>58.33%</td>
</tr>
<tr>
<td>wb_cyc</td>
<td>42</td>
<td>20</td>
<td>56.43%</td>
</tr>
<tr>
<td>wb_we</td>
<td>18</td>
<td>10</td>
<td>67.78%</td>
</tr>
<tr>
<td>wb_adr</td>
<td>68</td>
<td>13</td>
<td>62.22%</td>
</tr>
</tbody>
</table>

Figure 4.13: Coverage comparison between directed test and counterexample method on Rigel design.

pipeline designs are included, showing the coverage achieved by these two methods. The directed test suite from Rigel is written by the designers. Figure 4.13 shows comparisons to the GoldMine’s counterexample generated test method and directed test method, applied to different designs. It can be observed that our counterexample based test generation method can help directed test to continually improve coverage. For example, the condition coverage in fetch_stage_module is improved from 63.33% to 95.3%.

4.7 Practical Limitation to Achieve 100% Coverage

Theoretically, the counterexample-based incremental method can reach 100% input space coverage for all target designs. However, there are four implementation limitations which can decrease the maximum coverage attained. The first issue occurs if there is inconsistency of counterexample’s simulation
traces between the simulator and the formal verifier. Occasionally, the generated simulation traces by simulator do not follow the counterexample produced by formal verifier. The solution to this problem is to have the formal verifier directly produce the waveform of the counterexample. This ensures that the expected counterexample will always show up in the waveform. The disadvantage of this method is that the formal verifier must have the ability to directly produce a waveform, so free tools such as SMV cannot be used.

The second issue preventing full input space coverage is that the implementation only tries to generate sequential (temporal) assertions or combinational assertions. It does not specifically differentiate combinational output and sequential output. There are a couple of potential solutions to this problem. One is that the output can be treated as a separate combinational output and sequential output. This should result in full input space coverage for either the combinational version or the sequential version of the output.

The third implementation issue is that data mining used in GoldMine can be ineffective in certain designs leading to high runtime, more iterations and poor assertion quality. For example, outputs that represent data path tend to not have optimal results. This happens because individual bits are used in the data mining process and it is difficult to find relationships among a large number of bits. The solution to this problem may be to use a higher level abstraction instead of data mining the individual bits. We plan to resolve this issue in future work.

The final implementation issue is the limited capacity of the formal verification tool. Even though there are many well designed formal verification tools, the state space explosion problem can prevent the tool from giving a definite answer about the validity of an assertion. This is what Incisive Formal Verifier refers to as an explored assertion. The effect on GoldMine is that the explored assertion may not be a true assertion and should not be added to the solution set. In current implementation, the explored assertions must be treated as failed assertions that cannot be refined. The potential solution to this problem is using abstraction to reduce the complexity of model checking.

### 4.8 Discussion about Final Decision Tree

In GoldMine, when all assertions on decision tree are true assertion, the decision tree converges to final decision tree (FDT) and represents the function of target
output.

As an alternative to FDT, binary decision diagrams (BDD) [99] is widely adopted to compactly represent Boolean function. However, FDT is dynamically and incrementally constructed from concrete simulation data while BDD is statically built from logic function. This dynamic and incremental building characteristic gives rise to FDT several unique characteristics in this special context of assertion/test generation.

Due to the dynamic building characteristic, FDT is able to include the reachable states in the function of target output and each generated assertion can be triggered by at least one reachable state. In addition, as an implementation optimization, we can prune the subtree with all true assertions on leaf nodes during the incremental construction process of FDT. In this section, we first compare BDD and FDT in terms of reachable states computation and then give formal proofs of the FDT characteristic of including reachable states and finally describe the dynamic pruning process of FDT.

4.8.1 Reachable States of Sequential Design

For the target output in a sequential design, FDT actually reconstructs one logic function from dynamic simulation data. The primary input and state variables within the output’s logic cone consist of parameters of this logic function. Meanwhile, we can use BDD to represent the output’s logic function in design in terms of the same parameters. The difference between these two logic functions lies in the including of reachable states.

In sequential design, not all states are reachable from initial states. Using BDD to compute reachability states involves the fix point computation [100]. Unfortunately, this fixpoint computation always suffers from state space explosion. Without this fix point computation, simply using BDD to build the function of target output is not able to include the reachable states. If we view each path from root to terminal node in BDD as one assertion, some assertions may correspond to unreachable states and are thus vacuous. However, FDT is capable of automatically including the reachable states constraints (theorem 5) and all generated assertions are non-vacuous (theorem 6).

The advantage of FDT in this context comes from the using of dynamic simulation data and formal verifier. The decision tree partitions the simulation traces
based on splitting variables. Finally, each leaf node of FDT corresponds to at least one concrete and reachable states.

Given one target output \( z = f(x_1,x_2,...x_m,s_1,s_2...s_n) \) in sequential design, where \( x_1,x_2,...,x_m \) and \( s_1,s_2...s_n \) are input variables and state variables in the logic cone of \( z \), the set of reachable states of the design is represented as \( R(s_1,s_2 \ldots s_n) \), which can be calculated using fixpoint computation [5]. Note that we simply discard other state variables outside the logic cone of \( z \). Each satisfiable assignment of \( R(s_1,s_2 \ldots s_n) \) is a reachable state of the design. In symbolic model checking [5], the transition relation and initial state are all encoded into BDD and fixpoint computation is done based on BDD operation [101].

For the target output \( z \), we denote the represented function by FDT as \( z_{fdt}(x_1,x_2,...x_m,s_1,s_2 \ldots s_n) \). For each generated assertion, there are two kinds of consequent: \( z_{fdt}=0 \) and \( z_{fdt}=1 \). We denote the set of assertions with output equal to zero as \( A = \{A_1, A_2 \ldots A_p\} \) and the set of assertions with output equal to one as \( B = \{B_1, B_2 \ldots B_q\} \). We use the \textit{Ante} operator to represent the antecedent of assertion. The function \( z_{fdt} \) can then be defined as:

\[
z_{fdt}(x_1,x_2,...x_m,s_1,s_2 \ldots s_n) = \bigvee_{1 \leq r \leq q} \text{Ante}(B_r).
\]

**Theorem 5** \( \forall c \in \{0,1\}, (f(x_1,x_2,...x_m,s_1,s_2 \ldots s_n)=c) \land R(s_1,s_2 \ldots s_n) \Rightarrow (z_{fdt}(x_1,x_2,...x_m,s_1,s_2 \ldots s_n)=c) \).

**Proof intuition:** Given one group of concrete values on primary input and state variables, if the concrete state does not satisfy \( R(s_1,s_2 \ldots s_n) \), the theorem is true. If the concrete state satisfies \( R(s_1,s_2 \ldots s_n) \), what we should prove now is that \( f \) and \( z_{fdt} \) produce the same value \( c \). (1) For the given input and state satisfying \( \text{Ante}(B_r) \), the value \( c \) predicted by \( z_{fdt} \) is 1. If \( f \) compute \( c = 0 \) for the given input and state, \( B_r \) will be a spurious assertion on the design. This contradicts with the definition of FDT: all assertions are true. (2) For the given input and state not satisfying any \( \text{Ante}(B_r) \), then there exists one assertion \( A_t \) to predict the value \( c \) to 0. Similarly, If \( f \) compute \( c = 1 \) for the given input and state, \( A_t \) will be a spurious assertion on the design.

The theorem 5 implies that FDT precisely represents the primary output function. For each reachable state and any value on primary input, the FDT is able to produce the same value as the output function in the design. For unreachable state, the FDT is allowed to produce an arbitrary value. However, if we use BDD to represent \( f \),
BDD is required to produce exactly the same value as the circuit on any given state and primary inputs regardless of whether the state is reachable or not.

**Theorem 6** \( \forall i \in \{1, 2 \ldots p\}, \text{Ante}(A_i) \land R(s_1, s_2 \ldots s_n) \) is always satisfiable. Likewise, \( \forall i \in \{1, 2 \ldots q\}, \text{Ante}(B_i) \land R(s_1, s_2 \ldots s_n) \) is also always satisfiable.

Proof intuition: FDT is built from concrete simulation data. Each path from root to leaf node corresponds to a set of simulation data which means these simulation data will satisfy \( \text{Ante}(A_i) \) or \( \text{Ante}(B_i) \). On the other hand, since these concrete simulation data are from the computation of the design, they will satisfy \( R(s_1, s_2 \ldots s_n) \).

The theorem 6 implies that all assertions generated from final decision tree are non-vacuous. In other words, the antecedent of each assertion is able to cover at least one reachable state and each reachable state is able to trigger one assertion. Comparing with the BDD representation of \( f \), each path from root node to leaf node in BDD can be considered as an assertion for corresponding output. As a result, some generated assertions from BDD are vacuous since the antecedent of these assertions corresponds to unreachable state. If these assertions are output for verification, it will lower the assertion coverage.

### 4.8.2 Dynamic Subtree Pruning

As an implementation optimization, decision tree can prune subtree with all true assertions on leaf nodes during the incremental construction process. Since the incremental decision tree algorithm will stop splitting on the nodes with true assertions and only continue to split on spurious assertions, it is unnecessary to keep the true assertions subtree in memory. It is possible to apply a memory optimization strategy to dynamically prune the subtree with all true assertions in each refinement iteration step. In contrast to BDDs, this dynamical pruning effectively walks around the problem of memory explosion in this context. Moreover, removing the subtree with true assertions does not compromise the functionality of the corresponding output.
4.9 Related Work

Our counterexample based stimulus generation approach distinguishes itself from all the existing coverage guided test generation approaches in that the generated counterexample is able to automatically explore logic not covered by previous stimulus. Counterexample-based refinement of abstractions for verification has been studied widely [10]. The idea of generating tests from counterexamples using model checking has been explored in software testing and hardware validation [102]–[104]. These methods require a predefined set of properties and then formally verify these properties. In our work, the set of properties are generated automatically, minimizing human intervention in the loop. Many techniques in prior art automatically generate validation patterns by incorporating coverage feedback [105],[106] dynamically. However, they do not use a flow similar to GoldMine for generating feedback.

Statistical methods have been adopted in hardware validation for assertion generation [31],[33],[107] and test generation [103],[108],[109]. IODINE [31] tries to automatically infer likely invariants by hypothesizing a set of predefined invariant pattern across one or more variables in the design and then analyzing the design’s dynamic behavior during simulation. The generated assertions need not be sound, as well as they are usually simple assertions like one-hot encoding. Invariant generation in software verification has been approached in [110]–[112] to speedup model checking.

The similar idea of coverage closure or complete coverage has also been investigated by both academia and industry [113]–[116]. Three main differences exist between these works and ours: (1) Their methods always assume a predefined set of assertions. (2) Their methods do not utilize the dynamic simulation traces and are not able to exclude unreachable states. (3) Their methods do not build an automatic and incremental feedback loop.

In the field of data mining, incremental decision tree algorithms like VFDT (very fast decision trees) [117] are explored to allow an existing tree to be updated or revised using new data instances. These are typically applied to handling stream data whose characteristics change over time.
4.10 Conclusions

In conclusion, we have presented a completely automated stimulus generation methodology for systematic coverage closure based on GoldMine. The forward progress and termination properties of the algorithm make it a sound and practically attractive solution.
CHAPTER 5

WORD LEVEL FEATURE DISCOVERY TO ENHANCE QUALITY OF ASSERTION MINING

5.1 Introduction

Assertion based verification is an increasingly popular verification methodology [18]. Assertions are used in formal property checking as well as simulation based verification to monitor dynamic simulation, improve internal signal observability and reduce debug effort [18].

Current assertion generation solutions [28]–[30], [34] generate assertions at the bit level and the term-level information from the RTL abstractions is completely lost. Even if there are word-level variables in RTL, all bits are ungrouped for bit-level feature and target selection. As a result, all generated assertions are for every bit of the target variable (output), and the features are selected as RTL variables, one bit at a time.

These mechanically generated bit-level assertions have multiple disadvantages. Firstly, the assertions have low readability. Since each bit of the existing RTL word-level structure is treated as an independent variable in the learning engine, the generated assertions are typically not in a human-digestible form. Assertion based checking is typically used in the RTL phase, so decomposition of assertions into bit phase belies the purpose of automatic assertion generation. Frequently, designers find the machine generated data too difficult to parse and assimilate since it is at a lower level of abstraction. Secondly, each generated bit-level assertion has very low coverage of input space of target variable. Thirdly, the bit-level assertions tend to be repetitive, and therefore numerous. This is because a word-level relationship like \((a > b)\) where \(a\) and \(b\) are 16 bits wide, would be captured by 16 different bit-level assertions. These disadvantages drastically limit the usability of the mechanically generated bit-level assertions.

In this chapter, we present a technique that uses static and dynamic analysis of RTL code to discover word level features. The generated word level features, which
are in terms of primary inputs, are used by machine learning algorithm. This allows
the generated assertions to be at the same level of abstraction as RTL. All the anal-
ysis is done in steps preceding the machine learning algorithm phase. We do not
modify the learning algorithms themselves to achieve our goal. The machine learn-
ing algorithm, as such is agnostic to the level of abstraction of its features.

We identify conditional expressions in the RTL as initial word level predicates.
In order to obtain word level features, the conditional expression needs to be in
terms of primary inputs. This computation has been called the weakest precondition
computation [118] and has been used in software program analysis. We use this
methodology, adapted to RTL, to discover word level features.

The weakest precondition for a given predicate is computed from the RTL source
code. For temporal assertion generation, we require weakest precondition to be
computed for the length of temporal behavior (consecutive cycles) that we are
interested in. The resulting word level features will be in terms of primary input vari-
ables at every cycle within the given length and register variables (pseudo primary
inputs) in the first cycle.

Statically computing weakest precondition along all possible paths is subject to
blowup due to the inclusion of path conditions [53], [119]. Moreover, this static
computation method is unaware of infeasible paths. Hence, the generated word
level features may become arbitrarily complex. They will not serve the purpose of
increasing readability of our assertions. To avoid this blowup, we use concrete sim-
ulation to guide the weakest precondition computation along the feasible simulation
paths. The path conditions are not retained in predicates during the computation.
The resulting word level features are highly simplified. Therefore, our simulation
guided weakest precondition method combines static and dynamic analysis of RTL
code to discover word level features.

We further improve the generated assertions by post-processing them based on
design knowledge to remove redundancies in the assertions. Overlap in known ex-
clusive features can result in over-constrained assertions. For instance, the assertion
$(p_1 \land \neg p_2) \Rightarrow (sum\_sel = 1)^1$ can be reported by the learning algorithm, where
$(p_1 : opcode = ADD)$ and $(p_2 : opcode = SUB)$ are mutually exclusive. There-
fore, predicate $\neg p_2$ is redundant and can be removed.

The target can also be in the form of word level predicate. A word level variable
may have many possible values. We analyze RTL code to identify word level output

---

1The “=” symbol in this chapter is used to represent the comparison proposition that evaluates to
be true/false. Syntactic representations in C-like languages for this operator are “==”
variable with constant assignments. The word level variable along with its assigned constant is used as the word level target. Hence, the learning algorithm does not have to decipher each bit of the word level variables.

After discovering word level features or targets, any learning algorithm can be employed for assertion generation. We use a decision tree based learning algorithm for our experiments. We apply our word level assertion generation technique to Ethernet MAC, I²C, and OpenRISC designs [1]. We compare the generated word level assertions with bit level assertions. Using word level features or targets, fewer assertions are generated and the percentage of generated true assertions is higher. The average number of propositions in word level assertions is nearly 50% of that in bit level assertions. These measurements reflect the higher readability and expressiveness of generated word level assertions. Moreover, the word level assertions tend to be able to detect more injected bugs than bit level assertions.

5.2 Background

This section introduces the terms used in this chapter. We also provide a background on feature selection.

5.2.1 Definitions

We treat the RTL source code as a “program” as in [40]. Our static analysis is done on the control data flow graph (CDFG) [119] of the RTL design.

A target is a variable for which we want to generate assertions. Variables in the logic cone of a target are those variables that can affect the value of the target [40]. A feature is a variable that is used to predict the target’s value. The generated assertions are of form $A \Rightarrow B^2$, where the antecedent $A$ can be a temporal or propositional formula in terms of the features and the consequent $B$ is written as a temporal or propositional formula in terms of the target.

A word level variable refers to a variable with bit width larger than 1 in the RTL design. A signal described by a bit-vector is typically considered a word level variable.

A conditional expression in RTL is an expression evaluated to be true/false to determine which branch should be executed. For example: a case statement or an

\footnote{We use LTL [7] notation for expressing generated assertions in this chapter.}
if-else branch may include a conditional expression. If a conditional expression is in terms of word level variables, we refer to it as word level conditional expression.

A word level predicate is a first order formula in terms of word level variables and is evaluated to be true/false. Typically, a word level predicate can be a word level conditional expression in RTL.

A word level assertion is an assertion that has at least one word level predicate as a proposition in its antecedent or consequent.

The mining window length is the duration of time cycles for which we want the generated assertions to capture temporal behavior. It depends on the sequential depth of target signal.

A Use-Definition Chain (UD Chain) is a data structure consisting of a used variable and all the definitions of that variable that can reach that use without any other intervening definitions.

In software verification, weakest precondition, denoted by \( wp(st, P) \), is usually defined with respect to a predicate (postcondition) \( P \) and statement \( st \) [53], [118], [119]. \( wp \) is the weakest condition that is true before the execution of statement \( st \) and also guarantees to meet the postcondition \( P \) after the execution of \( st \).

Considering the truth table of a target’s function in terms of features, a table entry is covered by a given assertion if the concrete value of the entry can satisfy the antecedent of the given assertion. The input space coverage of a given assertion refers to the percentage of truth table entries covered by the assertion.

5.2.2 Feature Selection

In machine learning, feature selection refers to the selection of a subset of input variables by eliminating variables with little or no predictive information [120]. Given an RTL target variable for which want to generate assertions, the primary input variables in the logic cone of the target are selected as features. We only include primary input variables because the generated assertions will be able to cover all functions within the logic cone of target.

For temporal assertion generation, the design is unrolled for the same number of cycles as mining window length and the sequential variables are annotated with the cycle in which they are assigned. For each variable, a different cycle annotation is treated as different sequential variable. The sequential variables are then treated the same manner as combinational variables. As an example, we can generate the
temporal assertion: \( a \land X\neg b \Rightarrow XX(c = 1) \), where \( a \) and \( b \) are at different cycles.

Besides the primary input variables within mining window, the register/state variables on the first cycle in the target’s logic cone are also treated as primary inputs for feature selection. The first cycle’s register variable refers to the state variables in farthest back temporal stage within the mining window. Our generated assertions can cover the unrolled logic functions within the mining window.

5.3 A Motivating Example

(1) Verilog example and logic cone of target

```
1. module or1200_ctrl(clk,...);
2. input rst, clk, ex_freeze, id_freeze, flushpipe;
3. input [32:0] if_insn;
4. output [3:0] alu_op;
5. reg[32:0] id_insn;
6. reg[3:0] alu_op;
7. always@(posedge clk)
8. if(rst || flushpipe)
9. id_insn <= {6'h5, 26'h0410000};
10. else if(!id_freeze)
11. id_insn <= if_insn;
12. else
13. id_insn <= id_insn;
14. always@(posedge clk)
15. if(rst)
16. alu_op <= `ALU_NOP;
17. else if(ex_freeze&id_freeze|flushpipe)
18. alu_op <= `ALU_NOP;
19. else if(!ex_freeze)
20. case(id_insn[31:26])
21. `OR32_J: alu_op <= `ALU_IMM;
22. `OR32_ORI: alu_op <= `ALU_OR;
23. `OR32_ADDI: alu_op <= `ALU_ADD;
24. ______
25. endcase
26. else
27. alu_op <= `ALU_NOP;
28. endmodule
```

Figure 5.1: A motivating Verilog example [1] for a comparison between word level assertions and bit level assertions. The word level feature and the word level target are highlighted in the word level assertion. Reset signal \( rst \) is disabled in sample assertions. Mining window length is 2 for temporal assertion generation. The \( \text{Var}(\#) \) in the logic cone denotes the variable’s annotated cycle index.

In Figure 5.1, we show a simple Verilog example from the decoder module of
OR1200 [1] and the corresponding sample word and bit level assertions. We also show the logic cone for the target $alu_{op}[0]$. The $if_{insn}$ represents the instruction from instruction fetch module and the $id_{insn}$ represents the current instruction for decoding. The first always process (line 7-13) determines the instruction for decoding. The second always process (line 14-27) assigns values to $alu_{op}$, which determines the functionality of ALU.

We set the word level target as $alu_{op}[3 : 0] = 'ALU.OR$, where $alu_{op}[3 : 0]$ is a word level variable. We generate assertions capturing two cycles’ temporal behavior of the target. The bit level features in this example are: $id_{freeze}$ and $flushpipe$ in the first and the second cycles, $ex_{freeze}$ in the second cycle, and $if_{insn}[31 : 26]$ and $id_{insn}[31 : 26]$ in the first cycle. If we use word level features, $(if_{insn}[31 : 26] = 'OR32.ORI)$ and $(id_{insn}[31 : 26] = 'OR32.ORI)$ will be the discovered word level features.

In the example, we show sample bit level and word level assertions generated using the decision tree based learning algorithm (explained in section IV.C). The word level assertion states “if the opcode of the fetched instruction is $ORI$ in current cycle, the opcode of ALU will be $OR$ operation two cycles later.” It assumes that execution unit and decoder is not disabled and pipeline is not flushed.

We can see that the word level assertion is more readable than the bit level assertion. For example, it is hard to parse the meaning of single bit variables such as $if_{insn}[31], alu_{op}[3]$ in the bit level assertion. In addition, the input space coverage of the word level assertion is higher than that of the bit level assertion, because the number of features is reduced when using word level features. As a result, covering the entire input space requires significantly fewer assertions.

5.4 Our Procedure for Automatic Word Level Assertion Generation

In a typical flow for automatic assertion generation using machine learning [28], [29], [31], [34], an RTL design is simulated using random or directed tests and the simulation traces are passed as data to a machine learning engine. Bit level features and targets are selected. The learning algorithm then infers rules among features and targets from the data. Each rule corresponds to a candidate assertion for the target. Formal verification tool can be used to filter spurious assertions [29], [34].

We extend the flow to generate word level assertions in Figure 5.2. To generate
word level assertions, we require word level predicates to be provided as features or targets. Our extension, which is shown in the dotted block in Figure 5.2, is a preprocessing step for automatic assertion generation. Phase 1 in the flow discovers word level targets. Given a discovered word level target from phase 1 or bit level target, phase 2 is responsible for discovering word level features in the logic cone of the target. The discovered word level features and targets are instrumented in the RTL code. The updated RTL is simulated and the resulting traces are provided to the learning engine to generate candidate assertions.

![Diagram](image)

**Figure 5.2:** Our procedure for automatic word level assertion generation. Our contributions, which are shown in dotted block, focus how to automatically discover word level features and targets.

### 5.4.1 Phase 1: Discovering Word Level Targets

In Phase 1, we first identify the target we want to generate assertions for. Bit level outputs or word level predicates on outputs are set as targets for assertion generation. For word level targets, we consider bit-vector output variables with constant assignments in the RTL code.

In the bit level assertion generation, the target bit level variable’s value is deciphered by the machine learning algorithm itself. Bit level variables can have one
of two values: 0 and 1. Therefore the two propositions in the consequent for any bit level target variable \( t \) are \((t = 0)\) and \((t = 1)\). The machine learning algorithm thus deciphers the bit level predicate. However, at the word level, the variables are bit-vectors and can have many possible values. Deciphering all these values by a machine learning algorithm may lead to too many assertions, many of which could be spurious or irrelevant to the design. Hence, we provide the word level predicate itself as a target to the learning algorithm. In other words, the word level variable along with its intended value is given as a proposition. Consequently, the learning algorithm does not need to decipher the value of the word level variable.

We analyze all assignments to each word level output variable in RTL code. If all the assignments assign constant values to the word level output, we then produce the word level predicate as a target encoding whether the word level variable is equal to the assigned constant. In Figure 5.1, \( alu_{op} = 'ALU\_NOP' \), \( alu_{op} = 'ALU\_IMM' \), \( alu_{op} = 'ALU\_OR' \) and \( alu_{op} = 'ALU\_ADD' \) can all be word level targets.

5.4.2 Phase 2: Discovering Word Level Features

In Phase 2, we discover all the word level features that are in the logic cone of the target from Phase 1. Phase 2 has two subphases. The first subphase identifies all word level conditional expressions within the logic cone of the target from the RTL code. These expressions are set as word level predicates. However, they may not be in terms of primary inputs. Therefore, the second subphase uses a simulation guided weakest precondition computation to discover all word level features in terms of the primary inputs from the word level predicates. This phase will be elaborated in section 5.5.

It should be noted that the variable, which is in target’s logic cone but not used by any discovered word level feature, should also be output as feature. Moreover, if a word level variable is already used by a discovered word level feature, some bits of the variable may be selected as features. The reason is that our method is based on the simulation, which may fail to cover all potential features.
5.4.3 Data Generation and Learning Algorithm

After the discovery of the word level targets and features, every bit of the corresponding word level variable will be hidden from the learning algorithm. To get the concrete simulation values of these features and targets for learning engine, we instrument them back to RTL code and rerun the simulation. The new simulation traces are then provided to learning engine.

The machine learning algorithm tries to infer a logical relationship between the target and features from simulation traces. Our word level feature or target discovery approach is independent of the machine learning algorithm. We use a decision tree based learning algorithm from GoldMine [29].

![Diagram](image)

Figure 5.3: Data structures for weakest precondition computation. The data structures are used for logic cone identification and simulation guided weakest precondition computation. The bold arrow lines show the concrete paths during simulation.
5.5 Simulation Guided Weakest Precondition Computation to Discover Word Level Features

5.5.1 Representing RTL as CDFGs

In this section, we introduce the data structures used in the simulation guided weakest precondition computation to discover word level features. We first use a Verilog parser to transform Verilog design into CDFG. Figure 5.3 shows the CDFG of the motivating Verilog example in Figure 5.1. There are three kinds of nodes in a CDFG: a branch node (e.g., $b_1$) corresponds to a branch statement in RTL; an assignment node (e.g., $b_2$) corresponds to an assignment statement in RTL; a merge node (e.g., $b_6$) corresponds to the end of a branch.

The multiple-cycle path in RTL refers to a path that is executed across multiple cycles. The Verilog program is unrolled and the variables in each cycle are annotated with the corresponding cycle index. Each path corresponds to a set of assignment statements and conditional expressions. The Path condition for an assignment statement is a conjunction of all conditional expressions leading to the execution of that assignment statement on the path. The CDFG records multi-cycle paths during simulation. Figure 5.3 shows two concrete paths in cycle 1 and cycle 2. The concrete path in cycle 1 is $b_1 - b_3 - b_4 - b_6 - b_7$ in the first always process and $b_8 - b_{10} - b_{12} - b_{13} - b_{16} - b_{18}$ in the second always process. The concrete path in cycle 2 is $b_1 - b_3 - b_5 - b_6 - b_7$ in the first process and $b_8 - b_{10} - b_{12} - b_{13} - b_{15} - b_{17}$ in the second process. These paths are used to guide weakest precondition computation.

The UD chain of a variable points to all statements that assign it. The UD-chain are used to compute the weakest precondition and track the variables in the logic cone of the target. Figure 5.3 shows the UD-chain for variable $id_{insn}$ in $b_{13}$. Statements in $b_2$, $b_4$ and $b_5$ define this variable. Note that the non-blocking assignment (“<=”) in a clock triggered process means the assigned value is used in next cycle.

5.5.2 Weakest Precondition Computation in RTL

In the example shown in Figure 5.1, we assume the postcondition predicate is $id_{insn}[31 : 26] = 'OR32_ADDI$. We backward substitute the variables used
in postcondition with the definitions to these variables. There are three definitions in \(b_2, b_4\) and \(b_5\). We must simultaneously consider path conditions for the variables used in postcondition predicate. The resulting weakest precondition is computed as follows:

**Example (1): static weakest precondition computation**

\[
wp(T, id_{insn}[31 : 26] = 'OR32_ADDI) = ((rst \lor flushpipe) \\
\Rightarrow 6'h5 = 'OR32_ADDI) \\
\land (\neg(rst \lor flushpipe) \land (\neg id_freeze) \\
\Rightarrow if_{insn}[31 : 26] = 'OR32_ADDI) \\
\land (\neg(rst \lor flushpipe) \land (id_freeze) \\
\Rightarrow id_{insn}[31 : 26] = 'OR32_ADDI)
\]

We employ the RTL weakest precondition to derive word level features from the conditional expressions in RTL. To guarantee that the resulting word level features are in terms of primary inputs, we set \(k\) as the mining window length and set all word level conditional expressions as postcondition predicates. These conditional expressions are within both the logic cone of the given target and the mining window. If postcondition \(P\) is in cycle \(i\) within the mining window, the \(wp^{i-1}\) will be computed.

### 5.5.3 Simulation Guided \(wp\) Computation

Statically computing the weakest precondition generates very complex and unreadable predicates. Assignments to the same variable on different paths are considered in the weakest precondition computation. The path condition for each assignment is also included in the resulting expression. In example (1), the path condition for \(if_{insn}[31 : 26] = 'OR32_ADDI\) is \(\neg(rst \lor flushpipe) \land (\neg id_freeze)\).

In addition, the path conditions for different variables used in postcondition \(P\) may conflict. If multiple variables are used in the postcondition predicate, the path conditions for the assignments to these variables are conjunct. However, static \(wp\) computation is unaware of the satisfiability of such condition. In other words, the conjunct paths for different variables may be infeasible.

Finally, the number of static paths increases exponentially when we compute
wp^k for large k. We will transitively track the definitions to all variables used in
postcondition predicates until the primary inputs or constants are reached. The
resulting weakest precondition is easy to blowup. In example (1), if we want to
compute wp^1, we should find definitions to id_insn used in b5 since it is not in
terms of primary inputs. There are three definitions to it in previous cycle. As a
result, 9 paths are taken into account.

We use a dynamic simulation guided weakest precondition computation to re-
place the static computation. The RTL design is first simulated using either directed
or random tests. All concrete paths are recorded during the simulation. We limit
the backward substitution only along concrete simulation path. In this way, we can
disregard the path conditions in wp computation since there is only one assignment
to any variable used in postcondition P along the concrete simulation path.

In the example in Figure 5.3, the concrete simulation paths in cycles 1 and 2 are
shown. Given the postcondition predicate P: id_insn[32 : 26] = ‘OR32.ORI in
cycle 2, we want to use simulation guided method to discover word level features.
The definition to id_insn on the concrete path is in statement b4. Using substitution,
we can discover the word level feature: if_insn[32 : 26] = ‘OR32.ORI. We can
see that the discovered word level feature using simulation guided wp computation
is simple and readable.

We simulate RTL design using directed or random tests to guide the wp^i com-
putation. The simulation path may span over millions of cycles, which is much
larger than mining window length len. However, the concrete paths used in wp^i
computation should be at most len cycles. We resolve this problem by shifting the
mining window during the simulation. Initially, simulation cycle 1 to cycle len is
in mining window. Then cycle 2 to cycle len + 1 is the new mining window. In
this way, the mining window is shifted every simulation cycle. The concrete paths
in every mining window can be used to guide the wp^i computation.

We set the mining window length to 2 in Figure 5.3 and the word level tar-
get is alu_op = ‘ALU.OR. There are several conditional expressions within
both the mining window and the logic cone of alu_op = ‘ALU.OR. Only the
id_insn(2)[32 : 26] = ‘OR32.ORI (b13 and b16) is at word level. The remaining
conditional expressions flushpipe(1)(b1), flushpipe(2)(b10), id_freeze(1)(b3),
id_freeze(2)(b10), ex_freeze(2)(b12) are selected as bit level features. Recall that
wp^1(id_insn(2)[31 : 26] = ‘OR32.ORI = (if_insn(1)[31 : 26] = ‘OR32.ORI).
When the mining window shifts to simulation cycle 2, the word level predicates
in cycle 3 and cycle 2 are considered. We assume that the concrete path in cy-
cle 3 is the same as that in cycle 2. In this case, the definition in $b_5$ is used.

$$wp^1(id_{\text{insn}}(2)[31:26] = 'OR32\_ORI) = (id_{\text{insn}}(1)[31:26] = 'OR32\_ORI).$$

We can see that the discovered word level features do not suffer from the blow-up problem even if we increase $k$ in the $wp^k$ computation. It should be noted that the concrete cycle numbers express the relative cycle order within the mining window. They are replaced with the X operator if assertions are expressed in LTL.

The simulation, being inexhaustive, cannot exhaust all feasible paths reaching postcondition $P$. However, finding a complete set of predicates as features for mining is not required in the context of assertion generation. The mining of assertions is not trying to extract the complete function of the given target. In addition, our method cannot guarantee that the extracted word level features are in terms of every primary input within the target’s logic cone. In this situation, we simply treat each bit of the input variables as a bit level feature.

### 5.6 Removing Redundant Propositions

The word level features as generated in our technique may have a causal relationship between them. They may also be mutually exclusive in certain design contexts. The learning algorithm may produce overconstrained or meaningless assertions. For example, both $state[15:0] = S1$ and $state[15:0] = S2$ can be discovered as word level features. However, $(state[15:0] = S1) \land \neg(state[15:0] = S2)$ may appear in assertion’s antecedent. Obviously, proposition $\neg(state[15:0] = S2)$ is redundant.

Figure 5.4 shows an example of the identification of mutually exclusive features. $P1$, $P2$ and $P3$ are word level conditional expressions in the logic cone of the target and they are set as postconditions for weakest precondition computation. For $P3$, two word level predicates are produced by computing two concrete simulation paths: Path 1 and Path 2. The assignment to variable $Y$ is 1 in path 1 and the assignment to variable $Y$ is 2 in path 2. We can see that the two discovered word level predicates are mutually exclusive.

Our solution to remove redundant propositions is a post-processing of all generated assertions to check for mutually exclusive propositions. When using the simulation guided $wp$ computation to discover word level features, we identify all the word level conditional expressions in the logic cone of a given target. For each conditional expression set as a postcondition predicate, we group all the discov-
Figure 5.4: Identification of mutually exclusive features during feature discovery

We implemented the simulation guided weakest precondition computation for Verilog RTL. Our implementation reads Verilog code and builds the corresponding CDFGs. We use the VCS simulator to simulate a design. Our implementation interacts with VCS through the directed programming interface (DPI). All dynamic simulation paths are recorded in the CDFGs. The designs used for the experiments include Ethernet MAC, I²C and OpenRISC [1]. We use the provided regression testbenches to generate simulation traces for assertion generation. Our implementation uses a decision tree based learning algorithm to mine assertion as in GoldMine [29].

We choose those target signals that have word level predicates within their logic cone. In the case that there is no word level predicate within the logic cone, we
set all the bit level variables as features. The word level experiments and the bit level experiments use the same simulation data. Our implementation uses Cadence IFV as the formal verification engine to check the generated candidate assertions. All experiments were run on an Intel Core 2 Quad with 4GB of memory. Most generation processes complete within half an hour depending on the IFV runtime. The first experiment shows the word level feature discovery results for each target signal. The following experiments compare the readability and expressiveness of generated word level assertions and bit level assertions from the following perspectives:

1. Number of generated candidate assertions.
2. Percentage of true assertions.
3. Average number of propositions in assertion’s antecedent.
4. Input space coverage analysis of generated assertions.
5. Analyzing relationship between word level assertions and bit level assertions.
6. Injecting bugs in RTL and using the generated assertions to detect the injected bugs.

5.7.1 Word Level Feature Discovery Results Using Simulation Guided wp Computation

The first experiment evaluates the results of our word level feature discovery method. The target signal and the mining window length are determined by the user beforehand. In Table 5.1, we show the number of word level features discovered by using simulation guided method and also the number of bit level features. In addition, the number of exclusive features column shows the number of detected mutually exclusive features. In Table 5.1, we can see that the number of features are significantly reduced by using word level predicates. On average, there are 56% fewer word level features than bit level features. The only exception is the Wb_ack signal in Ethernet MAC. There are 7 features that are mutually exclusive for Wb_ack and redundant propositions can be removed in the generated word level assertions. These word level features are used for all following experiments and the exclusive features are used to remove redundant propositions in generated word level assertions.
Table 5.1: Results of our word level feature discovery method. Some bit variables, which are in logic cone but not in predicates, should also be included in features for word level assertion generation. The number of features can be reduced when using word level features.

<table>
<thead>
<tr>
<th>Target Signals</th>
<th>Window Length</th>
<th>Number of Word Level Features</th>
<th>Number of Exclusive Features</th>
<th>Number of Bit Level Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C-scl_oen</td>
<td>3</td>
<td>38</td>
<td>23</td>
<td>53</td>
</tr>
<tr>
<td>I2C-core_cmd</td>
<td>2</td>
<td>36</td>
<td>23</td>
<td>36*4</td>
</tr>
<tr>
<td>I2C-sda_oen</td>
<td>3</td>
<td>43</td>
<td>24</td>
<td>60</td>
</tr>
<tr>
<td>I2C-sto_cond</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>I2C-busy</td>
<td>4</td>
<td>12</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>I2C-clk_en</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>EMAC-WB_ack_o</td>
<td>3</td>
<td>18</td>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>EMAC-UnicastOK</td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>EMAC-SetPauseTimer</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>52</td>
</tr>
<tr>
<td>EMAC-LatCrcError</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>EMAC-Ini-Crc</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>EMAC-ReclenOK</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>OR-alu_op=OR</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>OR-alu_op=NOP</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>OR-sig_trap</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>24</td>
</tr>
</tbody>
</table>

5.7.2 Number of Generated Word Level Assertions and Bit Level Assertions

In this experiment, the assertions generated by using word level features are compared with those generated using bit level features. Figure 5.5 shows the number of candidate assertions generated by the two methods. We can observe that the number of assertions generated using word level features is fewer than the number using bit level features. Intuitively, the provided word level features prevent learning engine from generating too specific assertions.

5.7.3 Percentage of True Assertions in Candidate Assertions

Given the same simulation traces, we observe that the assertion generation using word level features is able to output a higher percentage of true assertions. The result is shown in Figure 5.6. The only exception is the sto_cond signal, in which some extracted predicates are not activated as frequently as bit level variables. In addition, the clk_en signal has no true assertions because the simulation traces do
not sufficiently cover this target’s function. As an extreme example, there is no true bit level assertion for ReceivedLengthOK. In the design, this target depends on the comparison of two 16-bit signals. Bit level features are not able to capture this comparison relationship and the machine learning algorithm is not clever enough to deduce this comparison relationship from simulation data. As a result, the generated assertions are all spurious.

It should be noted that we are not trying to improve the true assertion percentage. It can be improved by analyzing false assertions and then generating high coverage tests, since a false assertion implies a coverage hole in the simulation traces or a bug in the RTL design.

5.7.4 Average Number of Propositions in Assertion’s Antecedent

In this experiment, we compare the average number of propositions in the antecedent of the generated assertions. From Figure 5.7, it can be observed that the average number of propositions in the antecedent of assertions using word level features is nearly 50% fewer than the average number using bit level features in several cases. Bit level assertions use each bit as a feature and tend to be overconstrained.
5.7.5 Input Space Coverage Analysis of Generated Assertions

In this experiment, we compare the input space coverage of the word level and bit level assertions. We show that the input space coverage increases with the number of generated assertions. From Figure 5.8, we can observe that the input space coverage of the word level assertions increases more quickly with the number of the assertions when compared to the bit level assertions. In this example, the input space coverage of 5 word level assertions is nearly 90% while the input space coverage of the same number of bit level assertions is only 36%.

Table 5.2: We show that one word level assertion can cover multiple bit level assertions. We also show the used word level feature for generating word level assertions.

<table>
<thead>
<tr>
<th>Assertion ID</th>
<th>Word Level Features</th>
<th># Covered Bit Level Assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>((\text{Crc}[31 : 0] != 32'hc704dd76))</td>
<td>15</td>
</tr>
<tr>
<td>A2</td>
<td>((\text{NibCnt}[13 : 0] == 14))</td>
<td>14</td>
</tr>
<tr>
<td>A3</td>
<td>((\text{DlyCrcCnt}[3 : 0]); (\text{DlyCrcCnt}[3 : 0] &lt; 4'h9))</td>
<td>4</td>
</tr>
<tr>
<td>A4</td>
<td>((\text{ByteCnt}[15 : 0] == \text{MaxFL}[15 : 0]))</td>
<td>59</td>
</tr>
<tr>
<td>A5</td>
<td>((\text{RandomLatched} == 10'h0)))</td>
<td>4</td>
</tr>
</tbody>
</table>
5.7.6 Relationship between Word Level Assertions and Bit Level Assertions

In this experiment, we demonstrate that one word level assertion can cover several bit level assertions. This covering relationship means that the antecedent of bit level assertion implies the antecedent of word level assertion. Meanwhile, they assert the same value on target. For example: the word level assertion $a[7:0] > b[7:0] \Rightarrow (out = 1)$ covers bit level assertion $a[7] \land a[6] \land b[7] \land \neg b[6] \Rightarrow (out = 1)$ because $a[7] \land a[6] \land b[7] \land \neg b[6]$ implies $a[7:0] > b[7:0]$. In Table 5.2, we collect 5 word level assertions and analyze their covering relationship with the bit level assertions generated for the same target. It can be observed that one word level assertion can cover multiple bit level assertions. Intuitively, learning engine is not strong enough to derive the word level features from the simulation data. Providing word level features helps the learning engine to infer rules among word level variables.

5.7.7 Bug Detection Ability Comparison

This experiment uses assertions to detect bugs injected in the design. We want to demonstrate that a word level assertion is able to detect more bugs than its corresponding bit level assertion. We use a systematic mutation-based method to compare the assertions’ ability to detect bugs. The RTL code within the target’s logic
cone is mutated and then all the generated true assertions are formally checked on the mutated design. The failed assertions detect the corresponding bug in the mutated design. Four types of corner case bugs are injected: operator replacement, variable to constant replacement, constant replacement and relational operator replacement [98].

Table 5.3: The detecting of injected corner case bugs per word level assertion and bit level assertion. Word level assertions are able to detect more injected bugs.

<table>
<thead>
<tr>
<th>Target Signal</th>
<th>Injected Bugs</th>
<th>No. of Assertions</th>
<th>Bugs Detected Per Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Word</td>
<td>Bit</td>
</tr>
<tr>
<td>Initialize_crc</td>
<td>8</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>SetPauseTimer</td>
<td>11</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>sel_oen</td>
<td>18</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Each true assertion is able to detect multiple bugs. We find that each injected bug is detected by at least one word level assertion. However, the bit level assertions are not able to detect every bug. Table 5.3 shows the average number of bugs detected by each assertion. It can be observed that word level assertions are able to detect more injected corner case bugs than bit level assertions. Intuitively, a bit level assertion tends to be more specific and thus fails to detect some corner case bugs.
5.8 Related Work and Conclusion

We discuss work that is related to different aspects of our approach. Our word level assertion approach distinguishes itself from current assertion generation in that it uses word level predicates to help learning engine to generate high quality assertions.

Assertion generation from RTL has been approached in the context of static analysis [32], and dynamic analysis [27],[30],[31],[33],[121]. In [31], the solution tries to infer likely invariants by hypothesizing a set of predefined assertion templates in the design and then match the simulation traces to the template. They do not use data mining and also the generated assertions are typically low level invariants.

Our concept of word level assertions is also inspired by the predicate abstraction in software [122] and hardware [53],[119]. They mainly use the weakest precondition computation to find new predicates which are used to refine original abstract model. Our method uses concrete simulation to guide the discovery of word level features. Word level analysis in software [27] always predefines several invariant templates among word level variables to match the running data.

In conclusion, we presented a word level assertion generation methodology and the word level features are discovered by combining static and dynamic analysis. Experimental results demonstrate that word level assertions have better expressiveness and readability, and can detect more bugs in design.
CHAPTER 6

AUTOMATIC GENERATION OF SYSTEM LEVEL ASSERTIONS FROM TRANSACTION LEVEL MODELS

6.1 Introduction

Assertions were recently introduced for the verification of SystemC designs [37]. Cycle based SystemC designs are similar to RTL designs in that they are also based on a clock signal. Therefore, assertions for such kind of SystemC designs resemble RTL assertions. TLMs, which are at a higher level, deal with transactions instead of cycles and therefore may not have the concept of clock or cycle. TLMs use function calls for communication between different modules and events to trigger the communication actions. Therefore, the assertions for TLMs involve the communication actions and operating conditions for these communications [37].

Academia and industry have recently proposed several solutions to automate the RTL assertion generation process [29]. GoldMine [29] is a representative tool that mines assertions from an RTL design. Others tools like [28], [31], [110], [121] use template matching or static analysis method.

It is highly desirable to generate assertions at the system level. The main motivation is that system level assertions achieve wider perspective and facilitate analysis of the design early in the design cycle. Assertions can enhance understanding of the functionality and performance of the design at the system level. They can also serve as a succinct expression of the specification, or the “contract” expected by the system level to the RTL implementation. There are many use cases for system level assertions: regression testing, simulation monitoring, and debugging of system level models. In addition, the assertions generated at system level can be potentially included as an assertion library for RTL design verification. In GoldMine and other RTL assertion generation engines, there is a fear of reproducing RTL design bugs in the assertions. This situation can be countered with system level assertion generation technology. If the assertions are generated from an independent specification entity, like a system level TLM description, this greatly improves the
confidence in the generated assertions.

In this chapter, we first attempt to extend GoldMine to automatically generate high quality assertions from Transaction Level Models using data mining. We apply sequential pattern mining to generate assertions for abstract functional behaviors of TLMs in [35]. All function calls and events occurring during one simulation run of TLM designs were ordered by time to form a sequence. Sequential pattern mining would then search for frequently occurring ordered events/function calls as patterns from sequences [3]. We instrumented the TLMs and recorded all executed functional calls, triggered events, and their occurring time during simulation as sequences. The patterns generated by sequential pattern mining can be thought of as candidate TLM assertions. The generated assertions were expressed using linear temporal logic\(^1\) [7].

A good TLM assertion should express the data propagation relationships among function parameters and return values [37]. However, with sequential pattern mining, it is not possible to mine such relationships directly from the simulation traces containing concrete values of these parameters. For example, the sequence \(S_1: \text{read}\_\text{mem}1(100) \rightarrow \text{write}\_\text{mem}2(100)\) means that reading concrete data 100 from memory 1 is always followed by writing 100 to memory 2. The concrete value of the function parameter may be 200 in another simulation trace leading to another sequence \(S_2: \text{read}\_\text{mem}1(200) \rightarrow \text{write}\_\text{mem}2(200)\). Sequential mining may output \(S_1\) and \(S_2\) as distinct concrete assertions. Such multiple concrete assertions are too repetitive and contain no extra information. A more general form of this assertion is \(S: \text{read}\_\text{mem}1(A) \rightarrow \text{write}\_\text{mem}2(A)\). This is a symbolic version of the concrete assertions \(S_1\) and \(S_2\). We symbolized the concrete parameters and return values in simulation traces using symbolic execution [42]. For each concrete simulation path, symbolic execution evaluated the parameters or return values in terms of given symbolic inputs. For the above example, the symbolized parameters of \(\text{read}\_\text{mem}1()\) and \(\text{write}\_\text{mem}2()\) are the same in each simulation trace so that the concrete values 100 and 200 do not interfere with the generation of the assertion \(S\).

The sequential pattern mining algorithm we used in [35] searches for relevant function calls or events in the whole sequence. Typically, several thousands of function calls or events are coincidental, but not causal. However, they are identified by sequential mining as assertions. As a result, the number of generated

\(^{1}\)In this chapter, we use \(A \rightarrow B\) to represent the form of linear temporal logic: \(A \Rightarrow FB\)
TLM assertions will increase exponentially with the size of the corresponding sequence. Many of these generated assertions are spurious, irrelevant, and of low quality. Generating too many assertions also hampers the usability of the assertions by humans. It is difficult for users to sift through thousands of assertions and use them for verification.

The sequential pattern mining algorithm also suffers from scalability, which makes the mining intractable for long simulation traces. The algorithm always searches for frequent sequence incrementally. The frequent sequences in current iteration are used to form longer candidate sequence in next iteration. Consequently, the number of candidate sequences will grow rapidly and be combinatorially explosive. In a DMA design trace including only 64 events, sequential pattern mining generates more than 500,000 TLM assertions.

Finally, assertions in [35] capture functionality without timing specifications. The generated TLM assertions express the ordering relationship among the function calls or events in the design. In practice, TLMs are employed for performance evaluation. The assertions for TLMs should also be able to express performance specification. In previous assertion $S$, it asserts that $\text{read}_\text{mem1}(A)$ always precede $\text{write}_\text{mem2}(A)$, while the latency between $\text{read}_\text{mem1}(A)$ and $\text{write}_\text{mem2}(A)$ is not specified in $S$. From the perspective of performance evaluation, the latency between function calls or events is also interesting to verification engineers.

As a second attempt, we present a scalable algorithm to generate fewer and more focused, useful assertions than sequential pattern mining. We use episode mining to generate TLM assertions. An episode is a partially ordered sequence of events occurring together [123]. Users are required to specify a time window constraining how close the events are. The time window is then slid along the time axis of the mined sequence. An episode can possibly occur in multiple sliding windows. The number of sliding windows, in which an episode occurs, is the frequency of this episode. In our context, the episode can be a sequence of function calls or events in TLM design. The generated frequent episodes are then interpreted as TLM assertions.

Episode mining is able to generate a more compact set of TLM assertions than sequential pattern mining, and the generated TLM assertions have higher quality than those generated by the sequential pattern mining in [35]. The time window of episode mining constrains the search space of candidate episodes in a sequence. Only the function calls or events occurring within the time window can be used to form candidate episodes. In our context, two function calls or events, which
occur far away from each other during simulation, tend to have no potential cause-effect relationship and should not be correlated in a TLM assertion. Episode mining avoids the generation of such kinds of assertions. Therefore, episode mining produces a more compact set of TLM assertions, and also the quality of assertions is higher.

In addition, episode mining algorithm is much more scalable than sequential pattern mining and can be used for a large simulation trace. Episode mining employs time window to prune the search space during an incremental mining process. As a result, the number of generated candidate episodes will not explode in the process.

We also extend the purely functional assertions from [35] to capture both performance/timing as well as functional specifications. We enhance the generated TLM assertions by annotating them with quantitative real time parameters. The assertions can then express the latency between function calls or events in the design. For example, assertion $S$ in the previous example after annotating may become $\text{read\_mem1}(A) \rightarrow 2_{[2,10]}\text{write\_mem2}(A)$ [50], which means that $\text{write\_mem2}(A)$ will occur between 2 and 10 time units after the occurrence of $\text{read\_mem1}(A)$. The assertions with quantitative real time parameters are useful for performance analysis. For different test scenarios, the time parameters in TLM assertion are distinctive and always reflect the latency of data transmission. Users are able to quickly localize the root causes of performance bottleneck through these TLM assertions. For example, a user may discover that the memory read from module $A$ always takes very long time. In our implementation, we extract the real time parameter from the simulation trace for each generated assertion.

Due to the paucity of formal property verification engines at system level, we only candidate TLM assertions. We believe that there is value in presenting candidate assertions to users of TLM designs. Those generated TLM assertions are useful for design understanding and also help the users to understand the cause-effect relationship among the function calls or events. In addition, they can facilitate the debugging of TLMs.

We apply our TLM assertion generation technique to a transaction level AMBA based DMA controller and AXI based interconnection network platform in SystemC. We reuse AMBA based DMA controller design from [35] and additionally implement an AXI based interconnection network platform for our experiment. Since there is no available complex TLM platform for academic/research use, we

\[ A \Rightarrow F_{[a,b]} B \]
also release this platform in the public domain [124]. We demonstrate our TLM assertion generation on both designs and show that our assertions capture their specifications.

We evaluate system level assertions on the basis of standards we define for high quality assertions. We compare episode mining with sequential pattern mining, and show that the episode mining is more scalable and is able to generate a compact set of assertions for TLM verification. The number of assertions is reduced by 150 to 228 times. We also analyze the quality of TLM assertions generated from episode mining by measuring the distribution of time interval between the occurrence of two events/function calls in the generated assertions. The time interval of events/function calls is smaller than 50 time units in the assertion generated by episode mining, while this time interval in sequential pattern mining can be as great as the length of each sequence.

Based on our previous work [35], our new contributions are as follows. We generate much fewer assertion than our previous method in [35]. Assertions are more focused, and have fewer spurious or purely coincidental relationships. The used episode mining is scalable for design trace with arbitrary number of events, while the sequential pattern mining in [35] is only able to handle less than 64 events. We enhance the generated assertions with time annotation to express performance constraints. We finally demonstrate our method on realistic SystemC models.

6.2 Symbolic Execution of TLMs

As we introduced in Chapter 2, Symbolic execution refers to the execution of a single concrete path with symbolic inputs instead of concrete inputs [42]. It is used to reason about all the inputs that take the same path through a program and is applied in path based software testing. The symbolic execution follows the specified concrete path.

In the context of system level designs, the model is simulated with concrete inputs. The concrete path is recorded in instrumented variables, which indicates the taken branch during the concrete execution [44]. After that, the system level model is symbolically re-executed along the concrete path and all variables on the paths are evaluated using symbolic inputs. Although symbolic execution suffers from path explosion problem, we do not try to use symbolic execution to explore all paths in our context. Symbolic execution in our context is only along the concrete
simulation paths triggered by testbench. The symbolic expressions for function parameters and return values are then calculated on each concrete path. Therefore, the usage of symbolic execution in our context is scalable to large design.

<table>
<thead>
<tr>
<th>1</th>
<th>Source program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>dma_tb::run(){</td>
</tr>
<tr>
<td>2.</td>
<td>...</td>
</tr>
<tr>
<td>3.</td>
<td>src_addr=rand();</td>
</tr>
<tr>
<td>4.</td>
<td>value=src_addr+1024;</td>
</tr>
<tr>
<td>5.</td>
<td>dma1.write(SRC, value)</td>
</tr>
<tr>
<td>6.</td>
<td>...</td>
</tr>
<tr>
<td>7.</td>
<td>}</td>
</tr>
<tr>
<td>8.</td>
<td>dma::write(addr, data, ...){</td>
</tr>
<tr>
<td>9.</td>
<td>...</td>
</tr>
<tr>
<td>10.</td>
<td>case(addr)</td>
</tr>
<tr>
<td>11.</td>
<td>SRC:</td>
</tr>
<tr>
<td>12.</td>
<td>dma Src Addr=data;</td>
</tr>
<tr>
<td>13.</td>
<td>DST:</td>
</tr>
<tr>
<td>14.</td>
<td>...</td>
</tr>
<tr>
<td>15.</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>Concrete simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>dma_tb::run(){</td>
</tr>
<tr>
<td>2.</td>
<td>...</td>
</tr>
<tr>
<td>3.</td>
<td>src_addr=1038;</td>
</tr>
<tr>
<td>4.</td>
<td>value=2062;</td>
</tr>
<tr>
<td>5.</td>
<td>dma1.write(1, 2062)</td>
</tr>
<tr>
<td>6.</td>
<td>...</td>
</tr>
<tr>
<td>7.</td>
<td>}</td>
</tr>
<tr>
<td>8.</td>
<td>dma::write(1, 2062, ...){</td>
</tr>
<tr>
<td>9.</td>
<td>...</td>
</tr>
<tr>
<td>10.</td>
<td>case(addr)</td>
</tr>
<tr>
<td>11.</td>
<td>SRC:</td>
</tr>
<tr>
<td>12.</td>
<td>dma Src Addr=data;</td>
</tr>
<tr>
<td>13.</td>
<td>DST:</td>
</tr>
<tr>
<td>14.</td>
<td>...</td>
</tr>
<tr>
<td>15.</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>Symbolic execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>dma_tb::run(){</td>
</tr>
<tr>
<td>2.</td>
<td>...</td>
</tr>
<tr>
<td>3.</td>
<td>src addr=A;</td>
</tr>
<tr>
<td>4.</td>
<td>value=A+1024;</td>
</tr>
<tr>
<td>5.</td>
<td>dma1.write(1, A+1024)</td>
</tr>
<tr>
<td>6.</td>
<td>...</td>
</tr>
<tr>
<td>7.</td>
<td>}</td>
</tr>
<tr>
<td>8.</td>
<td>dma::write(1, A+1024, ...){</td>
</tr>
<tr>
<td>9.</td>
<td>...</td>
</tr>
<tr>
<td>10.</td>
<td>case(addr)</td>
</tr>
<tr>
<td>11.</td>
<td>SRC:</td>
</tr>
<tr>
<td>12.</td>
<td>dma Src Addr=A+1024;</td>
</tr>
<tr>
<td>13.</td>
<td>DST:</td>
</tr>
<tr>
<td>14.</td>
<td>...</td>
</tr>
<tr>
<td>15.</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 6.1: A simple program [2] and its corresponding concrete simulation and symbolic execution.

In Figure 6.1, we show a simple DMA testbench and its corresponding concrete simulation and symbolic execution. The call to `rand()` returns 1038 during concrete simulation/execution. The expressions are evaluated using concrete values and all variables are assigned concrete values during the execution. The symbolic execution, on the other hand, evaluates expressions symbolically along the concrete path using the initial symbolic input variables. The return value of function `rand()` is considered as a symbolic input variable `A`. Along the path of concrete simulation, the value is evaluated as `A + 1024`. The second parameter of the function `dma_write()` is evaluated as `A + 1024` in the path. It should be noted that `SRC` is a constant in the program and it is thus not evaluated with symbolic variables. In general, branch conditions along the path are extracted as path constraints during the symbolic execution.

### 6.3 TLM Assertion Definition

We express our generated TLM assertions in the form of temporal logic with quantitative real time constraints [50]. Our assertion definition considers both the quantitative real time constraint and function calls.

Let $F$ denote the set of functions in a TLM design $M$ and $E$ denote the set of
events in \( M \). In the reset of this chapter, we also view function call as a special event. We will use \( e \) to represent both \( f \) and \( e \).

An event set \( \Lambda \) is a set of events. In our context, any event in \( \Lambda \) is either a function call or an event in TLM designs. Formally, \( \Lambda = E \cup F \).

An event occurrence is a pair \((e, t)\), where \( e \in \Lambda \). \( t \) is the time when \( e \) occurs. It is also denoted as \( e_{\@ t} \).

An event sequence \( S \) on \( \Lambda \) is an ordered sequence of event occurrences which are ordered by the time \( t \) of each occurrence. \( S \) can be expressed as

\[
< (e_0, t_0), (e_1, t_1), \ldots, (e_n, t_n) >
\]

where \( \forall i \in [0, n - 1], t_i \leq t_{i+1} \) and \( e_i \in \Lambda \). The event sequence corresponds to TLM simulation trace.

The basic form of the generated TLM assertion is formally expressed as follows:

\[
e_1 \Rightarrow F_{[t_1,t_2]} e_2: \text{where} \ t_1 \geq 0 \ \text{and} \ t_2 \geq t_1
\]  

(6.1)

The generated TLM assertions belong to the safety properties of the TLM models. We omit the global \( G \) operator when expressing the assertion. Its semantics can be expressed as follows. During any execution/simulation, \( \forall e_1@t_x, \exists t_y \ s.t \ e_2@t_y \) and \( t_1 \leq t_y - t_x \leq t_2 \). It means when \( e_1 \) occurs at \( t_x \), \( e_2 \) will occur between \( t_x + t_1 \) and \( t_x + t_2 \). We also name \( e_1 \) the antecedent of the assertion and \( e_2 \) the consequent of the assertion.

If the antecedent of an assertion \( A_1 \) is the same as the consequence of an assertion \( A_2 \), we can concatenate them together to form a longer TLM assertions. For example, \( e_1 \Rightarrow F_{[t_1,t_2]} e_2 \) and \( e_2 \Rightarrow F_{[t_3,t_4]} e_3 \) can form a new assertion: \( e_1 \Rightarrow F_{[t_1,t_2]} e_2 \Rightarrow F_{[t_3,t_4]} e_3 \), which is also expressed as \( e_1 \rightarrow_{[t_1,t_2]} e_2 \rightarrow_{[t_3,t_4]} e_3 \) using our notation.

### 6.4 Flow of SystemC TLM Assertion Generation

Figure 6.2 shows the detailed framework for SystemC TLM assertion generation. The entire framework also includes formal verification of the generated TLM assertions and how to employ them for practical verification. In this chapter, we mainly focus on the assertion generation part which is enclosed in the dotted box.

Given a SystemC TLM design, we first instrument callback functions to record the parameter value and the time when the function is called, and the event occurrence [37]. The design is then simulated and simulation trace is recorded into a file. Data symbolization is applied to replace concrete values in simulation trace
Figure 6.2: Our vision of SystemC TLM assertion generation. The dotted line outlines the portion of the flow that we have implemented in this chapter. An important use case of our assertions can be as TLM assertions for SystemC model validation and debug or a reference library for RTL assertion generation.

with symbolic expressions. We attempt sequential mining and episode mining to discover frequently occurred patterns within the symbolized simulation trace. We annotate the patterns with quantitative real time parameters and output the generated TLM assertions. Finally, the generated TLM assertions are evaluated using our proposed standards.

These mined assertions can be formally checked on system-level design with commercial model checking tool. For example, the C model checking tool like BLAST or special SystemC verification tool [125] can be potentially applied for formal verification of SystemC design. The counterexamples can also be incorporated to refine the spurious assertions. Similarly, the feedback from design and verification engineer can also be used to direct the generation of new stimulus. Finally, all final true assertions can be output as a set of TLM assertions. To the best of our knowledge, verifying the generated TLM assertions is still an open problem and we do not try to solve it here. For employing the TLM assertions for RTL ver-
unication, the general method is to refine system level assertions to RTL assertions, which has been introduced in [126].

6.5 Data Generation

In the data generation stage, the target design simulated using several tests to generate trace for data mining. During the execution of each test, function calls and events triggering communication activity are recorded. At the entrance and exit of each communication function, code is manually instrumented to report the function call, parameters, and return value of that function call. Any event that occurs is reported as well. This reported data is piped to a file which will serve as the input for the data symbolization and episode mining algorithm. An example of this file can be seen in Figure 6.3.

- **Function calls**: Calls to, and returning from, functions always involve communication activity. The parameters passed to the function are also recorded. The return value of each function call is considered as a special function call.

- **Events**: These are instances of the built-in event type. Events are used to trigger the computation blocks or other communication actions. Therefore, they are indispensable components for expressing transaction level assertions.

We also record the exact time when the function calls or events occur during the simulation. The time is used in episode mining and also for determining the quantitative real time values of generated TLM assertions. It should be noted that a simulation trace consists of multiple runs of different tests. Sequential pattern mining in [35] generated frequent sequential patterns from these multiple simulation runs. In this chapter, we combine multiple runs into one simulation trace for episode mining. We will present in experiment section about how to combine multiple runs into one simulation trace for mining.

Our assertions are mined from the simulation traces. The quality of the applied test stimulus has a big impact on the generated assertions. If the test stimuli are not able to cover the entire design sufficiently, the generated assertions tend to be spurious assertions and are not able to capture the entire specifications of the design. Therefore, we require that the applied test stimulus should have high coverage of the design functionality. In practice, users can adopt real testcases from system
Figure 6.3: An example of one simulation run from a timed DMA controller design. The function `dma.write()` is a command called by DMA testbench which configures the controlling register in the DMA controller. `b_transport` is the primitive function call. `mem_read().return` is a function call return.

group due to the fast simulation speed at system level. Users can also adopt coverage metric to evaluate the test stimuli. We do not solve the problem of stimulus generation for TLMs here.

6.5.1 Data Symbolization Using Symbolic Execution

As we mention before, we require the TLM assertions to be able to capture the abstract behaviors of the design such as the data propagation relationship between different modules. A typical example of TLM assertion is shown as follows:

\[ TLM\text{ }Assertion: \text{ }tb.write(addr='SRC, data=src\_addr) \Rightarrow F[5,20] \text{ } mem1.read(addr=src\_addr, \ldots) \]

The `tb.write` and `mem1.read` are both function calls in SystemC TLM designs, and `addr` and `data` are parameters of the functions. It states that once the DMA testbench(`tb`) writes the address of the source memory to the `SRC` register in the DMA controller, the source memory(`mem1`) will issue a read operation with that address afterwards. It can be observed that the propagation of `src_addr` is incorporated into the assertion.

In the shown assertion example, the `data` parameter in function `tb.write` will be used as the `addr` parameter in the function `mem1.read`. During model simulation, the variable `src_addr` is assigned a different concrete value in each test. This means that if two function calls have differing parameters, they will be represented as unrelated items by the mining algorithm. This makes it difficult for the mining algorithm to realize that the same sequence occurs in each test since the functions

```
dma.write(p1=0, p2=12, p3=1)@100ns
mem_write_transfer()@110ns
mem_write_wait@110ns
mem_read_transfer@119ns
mem_read_wait@119ns
dma_write_done@400ns
m_irq_to_change@400ns
```

```
mem1.read(addr=src_addr, ...)
```
are seemingly unrelated.

To solve this problem, we use a method of data symbolization. Symbolic execution is employed to calculate the symbolic value of each parameter in terms of the symbolic input variables. The symbolic values, rather than the concrete values, are recorded for mining.

For transaction level designs, we specify all the design inputs as primary symbolic inputs in the top-level testbench module. In a constraint random testbench, the primary inputs are randomized during simulation. We specify a symbolic value for each primary input. The symbolic values are kept the same for each iteration of the test, although these inputs will be assigned different concrete values in each test during the model simulation. For example, in the test we used for the DMA controller, the source and destination address, along with the length of each transaction, were randomized. We specify different symbolic value for these input variables.

In each test simulation, the concrete execution path is tracked by recording the taken branch. At the end of the simulation, symbolic execution is initiated along the concrete execution path taken in the test. Each assignment expression on this path is calculated in terms of primary symbolic inputs. The function parameters and return values are replaced with expressions in terms of the primary symbolic values. These symbolic function calls replace the concrete function calls in the original trace. It should be noted that we do not need to symbolize the conditional expressions in each simulation path since the conditional expressions do not assign values to any variable in the model. In the example shown in Figure 6.1, it is shown that the \texttt{dma.write()} function is recorded as \texttt{dma.write(1, A + 1024)} rather than \texttt{dma.write(1, 2062)} in the simulation trace.

One symbolized trace is generated for each test simulation. All symbolized traces are collected and used as the database for assertion mining. In other words, the assertion generation is based on the traces of all simulation paths instead of a single path. Mining engine will discover the invariants among the traces of all simulation paths. The generated candidate assertions are checked on the whole design when using SystemC model checker.

6.6 Attempt I: Sequential Pattern Mining

Sequential pattern mining searches for frequent subsequences as patterns in a sequence data set, where a sequence records an ordering events [3]. The sequences
in the database are recorded with or without a concrete notion of time. A typical example of sequential pattern is “Customers who buy bread are likely to buy milk within one month.” For retail market, sequential patterns are useful for shelf placement and promotions. They also find applications in web access analysis and network intrusion detection.

<table>
<thead>
<tr>
<th>Seq ID</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;a(abc)(ac)d(cf)&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;(ad)c(bc)(ae)&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&lt;(ef)(ab)(df)gb&gt;</td>
</tr>
<tr>
<td>4</td>
<td>&lt;eg(al)cbo&gt;</td>
</tr>
</tbody>
</table>

Figure 6.4: A sequence database [3].

Figure 6.4 shows a sequence database and there are four sequences. In sequence 1, there are 5 events: (a),(abc),(ac),(d),(cf). In each event, there may be several items and these items are not ordered within event. An item can occur at most once in an event of a sequence and can occur multiple times in different events of a sequence. A sequence $\alpha$ is called a subsequence of another sequence $\beta$ if $\alpha$ can be derived from $\beta$ by deleting some items or events without changing the event order. For example, $<a(bc)dc>$ is a subsequence of $<a(abc)(ac)d(cf)>$. The support of a sequence $\alpha$ in a sequence database is the number of tuples in the database containing $\alpha$. A frequent sequence $\alpha$ is called a sequential pattern if the support of $\alpha$ is not less than a given minimum support threshold. The support parameter is the most widely used measure for evaluating sequential patterns and it denotes the frequency with which a pattern occurs. Consider the subsequence $s=<(ab)c>$, which is highlighted in the Figure 6.4. It occurs in sequence 1 and sequence 2. Therefore, the support of subsequence $s$ is 2. If the given minimum support threshold is less than 2, then sequence $s$ is frequent pattern.

The sequential mining algorithms are mostly based on the Apriori algorithm. In the first pass, it counts all frequent single items. These frequent items are then used to form candidate sequences with length 2. Similarly, all frequent sequences with length 2 are used to generate candidate sequences with length 3. This process is converged until no more frequent sequences are generated.
6.7 Attempt II: Episode Mining

In this section, we first give several basic definitions which are necessary to explain episode mining algorithm. Then we introduce the episode mining algorithm for generating frequent episodes from the TLM simulation traces.

6.7.1 Basic Definitions

An episode $\xi$ is an ordered sequence of events. The events in the episode must occur in the order denoted as $\preceq$. For example, if $e_1$ occurs before $e_2$ and $e_2$ occurs before $e_3$ in TLM design $M$, then $\xi$ can be $e_1 \preceq e_2 \preceq e_3$. We can generate subepisode of an episode by removing one or more events from that episode. $e_2 \preceq e_3$ is a subepisode of $\xi$. The prefix of an episode $\xi$ is a special subepisode by removing the last event of $\xi$. The postfix of an episode $\xi$ is a special subepisode by removing the first event of $\xi$. In previous episode $\xi$, $e_2 \preceq e_3$ is the postfix of $\xi$, and $e_1 \preceq e_2$ is the prefix of $\xi$.

A window constraint is a real value $w$, that refers to the width of a time window. We can slide the time window along the time axis of an event sequence $S$. At time $t_x$, the sliding window corresponds to a time interval $[t_x, t_x + w]$.

Given a window constraint $w$, an episode $\xi$ occurs in a window $[t_x, t_x + w]$ if $\forall e_m \preceq e_n$ in $\xi$, $\exists t_m, t_n$ such that $e_m@t_m$ and $e_n@t_n$ in event sequence and $t_x \leq t_m \leq t_n \leq (t_x + w)$. This means that every event in $\xi$ must occur in the sliding window $[t_x, t_x + w]$.

An episode could occur in multiple sliding windows as the window is slid along the time axis. We define the support of an episode as the number of sliding windows this episode could occur in.

Given a support threshold value, a frequent episode is one whose support value is larger than the given threshold. Given an episode $\xi$, if the support of its prefix is the same as the support of $\xi$, we say that the confidence of $\xi$ is 100%.

Figure 6.5 shows an episode example and an event sequence. The episode $e_1 \preceq e_2$ occurs three times in the given event sequence. The support is thus 3. $e_1$ is a subepisode of $e_1 \preceq e_2$ and $e_1$ is also a frequent episode.
Figure 6.5: A frequent episode example of an event sequence. The window constraint is 3.5 in this example and frequency threshold is 3.

6.7.2 Episode Mining Algorithm

Given an event set $E$, an event sequence $E_{Seq}$, a support threshold $Min\_supp$ and a window constraint $Win$, the episode mining algorithm tries to discover all frequent episodes from $S$. The algorithm is an incremental extension process. It first generates frequent episodes including only one event. Then it generates frequent two-event episodes from these frequent one-event episodes. Iteratively, it generates frequent episodes from the frequent episodes reported in last iteration until there are no more frequent episodes. The algorithm is based on the following observation: If an episode is frequent in an event sequence, then any subepisodes are also frequent [123].

The algorithm is shown in Algorithm 1. The input parameters are event set, event sequence, support threshold, and the window constraint. $C_{i+1}$ records candidate episodes generated from frequent episodes of previous iteration. $L_{i+1}$ records the frequent episodes in iteration $i$. $Freq\_Check$ function is responsible for checking whether the provided episode is frequent or not in the event sequence. The checking process needs to slide the window to count the support of each episode.

In every iteration, the algorithm generates candidate episodes first and then checks whether the candidate episodes are frequent or not in event sequence. If a candidate episode is frequent, it will be reported as a frequent episode and is also kept for generating episodes in the next iteration.

$Cand\_Gen$ function generates candidate episodes in current iteration $i$. The generation process is shown in Figure 6.6. The frequent episodes generated in iteration $i - 1$ are used to form candidate episodes for iteration $i$. All infrequent episodes in iteration $i - 1$ are discarded in the next iteration. $e_1 \preccurlyeq e_2$ in this example is not a
Algorithm 1 Episode Mining algorithm  

$\text{EpisodeMine}(E, E_{\text{Seq}}, \text{Min\_supp}, \text{Win})$  

1: $\text{FreqEpisode} = \emptyset$;  
2: $L_1 = \text{Freq\_Check}(E, \text{Min\_supp}, \text{Win}); L_2 = L_1$;  
3: $\textbf{for} (i = 1; L_{i+1} \neq \emptyset; i++) \textbf{do}$  
4: \hspace{0.5cm} $C_{i+1} = \text{Cand\_Gen}(L_i)$;  
5: \hspace{0.5cm} $L_{i+1} = \text{Freq\_Check}(C_{i+1}, \text{Min\_supp}, \text{Win})$;  
6: \hspace{0.5cm} $\text{FreqEpisode} = \text{FreqEpisode} \cup L_{i+1}$;  
7: $\textbf{end for}$  
8: $\text{return } \text{FreqEpisode}$;

frequent episode and is not used in the next iteration. Given two frequent episodes $\xi_1$ and $\xi_2$ in iteration $i$, we consider the prefix and postfix of both $\xi_1$ and $\xi_2$. If the prefix (postfix) of $\xi_1$ and the postfix (prefix) of $\xi_2$ are the same, $\xi_1$ and $\xi_2$ can form a candidate episode in next iteration. For example, $e_1 \preceq e_2 \preceq e_3$ and $e_2 \preceq e_3 \preceq e_4$ can be used to form candidate episode $e_1 \preceq e_2 \preceq e_3 \preceq e_4$.

During candidate generation, we also require that the generated episode is 100% confidence in the event trace since these assertions reflect the specification of the TLM design and they will be used for design debugging.

The generated frequent episodes are simply interpreted as TLM assertions. The order $\preceq$ in episode is translated to $\rightarrow$ in TLM assertion. For example, episode $e_1 \preceq e_2 \preceq e_3 \preceq e_4$ is translated into assertion $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4$.

Figure 6.6: The incremental candidate episode generation in episode mining. The algorithm incrementally generate candidate episodes with $i + 1$ events from frequent episodes with $i$ events.
6.8 Comparison between Sequential Pattern Mining and Episode Mining for TLM Assertion Generation

We first attempt sequential pattern mining to generate TLM assertions [35]. We found that it is inefficient when applying it for large simulation traces. Episode mining uses a window constraint to prune the search space and improves the scalability of mining algorithm. As shown in Figure 6.5, episode \( e_3 \preceq e_4 \) does not satisfy the window constraint and are not considered as candidate episodes in mining process.

The framework of mining frequent episodes was first proposed in [123]. It is different from sequential pattern mining. First, the episodes are mined from an event sequence in which events are ordered by their occurrence time. Sequential pattern mining discovers frequent subsequence from multiple event sequences. Second, the events of an episode must occur close enough in time. Therefore, episode mining employs a time window constraint to search for the candidate episodes. Sequential pattern mining only considers the order of events and searches for frequent patterns in each entire sequence. Finally, in sequential pattern mining, multiple occurrences of a pattern in one sequence is considered as one occurrence when calculating the support value. The episode mining calculates the number of occurrence of an episode within on sequence.

From the perspective of practical TLM specification, using a window to limit the search space is also meaningful. For example, DMA controller initiates two independent transactions. However, the function call or events occurring during the first transaction is not correlated to the function calls or events in the second transaction. Window constraint is able to avoid the generation of such assertions.

6.9 Quantitative Time Annotation

Given a frequent episode (assertion) reported by the mining algorithm, we need to annotate it with quantitative real time constraint. We check every adjacent event pair \(<e_i, e_j>\) in the episode (assertion) and extract the time parameters from event sequence. We initially set a lower bound as \(+\infty\) and upper bound as 0. For every episode occurrence \(<...e_i@t_i, e_j@t_j...>\) in event sequence, we update the lower bound to \(|t_j - t_i|\) if \(|t_j - t_i|\) is less than the current lower bound. Similarly, if the \(|t_j - t_i|\) is larger than upper bound, we update the upper bound accordingly.
6.10 Evaluation of TLM Assertions

In order to classify and evaluate the quality of generated assertions for transaction level designs, we propose the following standards for good TLM assertions:

- **Standard I**: If the assertion involves several function calls, the parameters of these function calls should be correlated and be able to express the data propagation property during the test. For example, \( f(A, B) \rightarrow g(C, B + 2) \).

- **Standard II**: The involved function calls or events in the assertion should be at different interfaces of the design. In other words, the assertion should be across at least one computation module. This standard is different from cycle based design’s assertions, in which we assert the relative timing of the signals in the same interface. In TLMs, the signals on the same interface are encapsulated in one function call.

- **Standard III**: The length of generated assertions should be constrained. We use the constraint 3 to limit the assertion length. For example, the assertion \( A \rightarrow B \rightarrow C \) has length 3. It will be very difficult to accurately localize the failed stage if an assertion is very long. Moreover, too long assertions will degrade the simulation performance if we will monitor the generated assertion during simulation.

6.11 TLM Benchmark Platform: An AXI Based Interconnection Network

To evaluate our research in system level, we specially implement a practical platform using SystemC TLM. We use a hierarchical AXI bus model as the interconnect. We model it at approximately-timed transaction level. Each transaction communication has multiple phases, and delays are annotated on process interactions. The model is shown in Figure 6.7, and the source code can be downloaded from [124]. The model includes one processor cluster and one DSP cluster, both of which serve as initiators. The platform is close to a real industrial system in wireless baseband application and is agreed upon as a common platform for experiment by our industry collaborators. Any CPU/DSP initiator is allowed to communicate with any target. We generated tests in five initiators to test the whole platform. Target 0 and target 1 are DMA controller. Target 2 and target 3 are eight bank memory
models, and they are able to respond to the requests from different memory banks in parallel.

Figure 6.7: Figure showing the framework of AXI based interconnection network. All interconnection buses are AXI.

6.12 Experimental Analysis

We apply our flow to generate system level assertions for the SystemC TLM designs. We use two SystemC designs for evaluating our proposed method. The first design is a transaction level AMBA-based DMA controller. The second design is the AXI based interconnection network benchmark. For the AMBA-based DMA controller design, we generate TLM assertions using our mining method and evaluate sample assertions with our proposed standards. We also compare the performance of episode mining with that of general sequential pattern mining [35]. For the AXI-based interconnection network design, we use our method to generate TLM assertions with annotated time constraints and analyze these assertions.

6.12.1 AMBA-based DMA Controller

The SystemC code in this experiment is from the DMA example of AMBA-PV API provided by ARM [2]. The entire environment consists of: 1) a simple testbench to program the DMA transfers; 2) an AMBA-PV bus decoder routing transactions between the system components; 3) a simple DMA controller model implementing a producer-consumer scheme; 4) two AMBA-PV memories. The DMA controller is responsible for transferring the data between memories according to DMA control
command. Each transfer is considered as one transaction. The testbench config-ures the DMA controller by writing control registers of DMA through AMBA-PV channel. The environment framework is shown in Figure 6.8. We can configure the DMA controller for multiple simulation runs.

![Figure 6.8: The framework of a transaction level AMBA-based DMA controller.](image)

Table 6.1: Evaluation of assertions generated by episode mining for a transaction level model of a DMA controller. Quantitative time constraints are discarded in the assertions since the DMA controller model is a programmer view model, and there is no timing information.

<table>
<thead>
<tr>
<th>Assertion</th>
<th>TLMs sample assertions</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>dma1.write(DST, B) (\rightarrow) mem2.write(B)</td>
<td>I+II+III</td>
</tr>
<tr>
<td>A2</td>
<td>tb.write(DST, B) (\rightarrow) dma1.write(DST, B)</td>
<td>I+III</td>
</tr>
<tr>
<td>A3</td>
<td>tb.write(CTRL, 0x01) (\rightarrow) m_start_transfer</td>
<td>II+III</td>
</tr>
<tr>
<td>A4</td>
<td>m_start_transfer (\rightarrow) dma_irq (\rightarrow) m_end_transfer</td>
<td>II+III</td>
</tr>
<tr>
<td>A5</td>
<td>burst_read(A, G) (\rightarrow) burst_write(B, G)</td>
<td>I+II+III</td>
</tr>
<tr>
<td>A6</td>
<td>tb.write(SRC, A) (\rightarrow) b_tran_rd(A, 0, H)</td>
<td>I+II+III</td>
</tr>
</tbody>
</table>

Augmenting DMA Controller with Timing

The DMA controller design we use is an untimed model for early software development, and there is no timing information in the model. All events/function calls occur at time zero during simulation and the delay between events/function calls is abstracted away for fast simulation. Absence of timing information in the untimed model does not mean that all events and function calls occur simultaneously.
Our TLM simulation trace still records the occurrence order of the function calls and events. In order to apply episode mining, we need to identify the time when a function call or an event occurs.

We need to preprocess the simulation trace of the untimed DMA controller model for episode mining. Recall that multiple simulation runs need to be combined into one simulation trace. We preprocess multiple simulation runs one at a time. For the first simulation run, we assign an ordering numerical value to each event or function calls. The numerical value initially starts at 0 and increments the order number by one for each function call or event within the first simulation run. The numerical value represents the artificial occurrence time of the event or function call within simulation run. When it comes to the end of one simulation run, we increment the ordering numerical value by the value of window constraint, and then continue to assign ordering numerical value for another simulation run. We thus avoid the interference between events/function calls in two different simulation runs. We wish to keep the events/function calls in two simulation runs independent while generating TLM assertions. Hence, we manually assign a sufficient gap between different simulation runs.

Evaluation of Generated Sample TLM Assertions

Table 6.1 shows several sample assertions automatically generated from the simulation traces. We also evaluate these assertions using our proposed standards for good assertions. The variables $A$, $B$, etc., in function call parameters are the symbolic input variables in the design. Table 6.2 describes what each assertion means.

Since the transaction level model focuses on communication between different modules, only the communication specification of the DMA controller is captured by the assertions. We also analyze the relationship between the generated assertions and the coverage of communication actions in the DMA controller.

- Configuration interface of DMA controller: The testbench serves as the AMBA master device and DMA controller is the slave device. The testbench issues various configuration command to the master interface and the DMA controller accepts the configuration. This functionality is covered by the sample assertions $A_2$, $A_3$ and $A_6$. 
- Communication between DMA controller and memories: The DMA controller serves as AMBA master and memory 1 and 2 are slaves. DMA controller reads the data from memory 1 and writes it to memory 2. This function is covered by A1 and A5.

- Interrupt interface between DMA controller and testbench: When the DMA controller finishes one transaction, it will set the interrupt request interface. This function is covered by A4.

Table 6.2: Functional descriptions of the sample set of assertions shown in Table 6.1. Our techniques are able to generate assertions which capture communication specification intent and temporal functionality.

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Once the dma testbench(t) writes the address of source memory to the DST register in DMA controller, the source memory(mem2) will issue a write operation with the address.</td>
</tr>
<tr>
<td>A2</td>
<td>The dma write function call follows the testbench write function call and they have the same function parameters.</td>
</tr>
<tr>
<td>A3</td>
<td>Once the testbench writes 0x01 to the control register, the event m_start_transfer will be issued.</td>
</tr>
<tr>
<td>A4</td>
<td>The three ordering events appear in every DMA transaction.</td>
</tr>
<tr>
<td>A5</td>
<td>The burst write function to destination memory follows the burst read function from the source memory and the length parameter is the same.</td>
</tr>
<tr>
<td>A6</td>
<td>Once the dma testbench writes the address of source memory to the SRC register in DMA controller, the b_transport function is called with the source address as first parameter. Function b_transport is an API function of the library.</td>
</tr>
</tbody>
</table>
Comparison between Episode Mining and General Sequential Pattern Mining

In this experiment, we compare the number of generated assertions and running time between episode mining and general sequential pattern mining as described in [35]. The time of simulation, instrumentation and symbolic execution is negligible when compared to the pattern mining time. For episode mining, we use one simulation trace consisting of multiple simulation runs and preprocess it by including artificial time for each event or function call. For sequential pattern mining, we use multiple simulation runs as in [35].

Table 6.3: Comparison between episode mining and general sequential pattern mining for TLMs assertions generation on DMA controller model. The number of generated assertions and running time are shown in the table. We also compare the average number of generated TLM assertions per event or function call in the design.

<table>
<thead>
<tr>
<th>Num of runs</th>
<th>Num. of Assertions</th>
<th>Num. of Assertions/Event</th>
<th>Run time</th>
<th>Num. of Assertions</th>
<th>Num. of Assertions/Event</th>
<th>Run time</th>
<th>Reduction in Num. of Assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8682</td>
<td>0.365s</td>
<td>130</td>
<td>1980363</td>
<td>10s</td>
<td>39943</td>
<td>228x</td>
</tr>
<tr>
<td>10</td>
<td>3923</td>
<td>0.292s</td>
<td>50</td>
<td>589823</td>
<td>3s</td>
<td>9216</td>
<td>150x</td>
</tr>
<tr>
<td>15</td>
<td>3923</td>
<td>0.219s</td>
<td>47</td>
<td>589823</td>
<td>4s</td>
<td>9216</td>
<td>150x</td>
</tr>
<tr>
<td>20</td>
<td>3923</td>
<td>0.243s</td>
<td>47</td>
<td>589823</td>
<td>8s</td>
<td>9216</td>
<td>150x</td>
</tr>
<tr>
<td>30</td>
<td>3923</td>
<td>0.233s</td>
<td>47</td>
<td>589823</td>
<td>5s</td>
<td>9216</td>
<td>150x</td>
</tr>
<tr>
<td>50</td>
<td>3923</td>
<td>0.363s</td>
<td>47</td>
<td>589823</td>
<td>6s</td>
<td>9216</td>
<td>150x</td>
</tr>
</tbody>
</table>

In Table 6.3, it can be observed that episode pattern mining is able to generate a much more compact set of assertions within 2 seconds than sequential pattern mining. For episode mining, we also do not constrain the number of events/function calls in the trace. However, for general sequential pattern mining, we include only 64 events/function calls in each simulation run since SPAM [127] does not support more than 64 transactions in our experiment. We discover that the number of assertions generated by sequential pattern mining is significantly larger than that generated by episode mining. From the table, the number of generated assertions by sequential pattern mining can be 150 to 228 times more than that by episode mining. Also, sequential pattern mining generates much more assertions per event or function call than episode mining does. As a result, it is also very difficult to find readable and useful assertions for TLM verification.

In Table 6.4, we show several sample assertions which are generated by sequential pattern mining but not generated by episode mining. It can be observed that there is no cause-effect relationship between the events within each assertion. For example, two DMA write function calls are included in assertion $Seq\_A1$. The
Table 6.4: Evaluation the quality of assertions generated by sequential pattern mining. The events within each assertion have no cause-effect relationship, and they are related coincidentally by the sequential mining algorithm. Episode mining, however, is able to avoid the generation of these low quality assertions.

<table>
<thead>
<tr>
<th>Assertion</th>
<th>TLMs sample assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq_A1</td>
<td>dma1.write(DST, B) → dma1.write(CTR, E)</td>
</tr>
<tr>
<td>Seq_A2</td>
<td>dma1.write(DST, B) → dma1.write(DST, B)</td>
</tr>
<tr>
<td>Seq_A3</td>
<td>tb.write(LEN, B) → m_end_transfer</td>
</tr>
</tbody>
</table>

first call writes the destination address, and the second one writes the control register to clear the interrupt after all transactions are finished. We cannot assert any relationship between these two function calls. Seq_A2 relates the same function call in two different transaction activity. The second write belongs to another independent transaction trace. There should be no cause-effect relationship between them. Sequential pattern mining does not take into account the interval between any events. It searches for the correlated events in the entire sequence. As a result, a lot of events are related coincidentally. This is also why sequential pattern mining algorithm generates much more assertions than episode mining does.

Figure 6.9: Figure showing the distribution of the time interval between two events/function calls of two-event assertions generated by episode mining. We fix one event/function call (write_source_addr) in the DMA controller and consider all assertions including this event.

We further analyze the quality of assertions by measuring the time interval between occurrence of events/function calls in the generated assertions. If the time interval between two events/function calls occurrence exceeds the lifetime of a transaction, then we conclude that there is no cause-effect relationship between these two events/function calls. This is reasonable, since an event in one transaction is unlikely to affect an event in another transaction. We find that episode mining tends
to correlate the events/function calls within small enough time interval. Figure 6.9 shows the distribution of the occurrence time interval for the assertions generated using episode mining. We used a window constraint of 100 time units. It can be observed that the time intervals between two events/function calls in each generated assertion is smaller than 50 time units. We cannot show the results for sequential pattern mining, because the tool [127] we used can only support a maximum of 64 events. This is an artificial constraints of the tool. Other implementation of sequential pattern mining might not impose such constraints. It is thus not feasible to compare it with episode mining that does not limit the number of events. In general, the sequential pattern mining’s time interval can be as great as the length of each sequence. In addition, there are hundred thousand assertions, and it is difficult to characterize the distribution of their time intervals.

### 6.12.2 AXI Based Interconnection Network Platform

In this section, we evaluate the assertions generated by episode mining for our AXI based interconnection network platform.

#### Table 6.5: Evaluation of assertions generated by episode mining for an AXI based interconnection network. The unit of time constraint is nanosecond. The used window constraint is 300ns.

<table>
<thead>
<tr>
<th>Assertion</th>
<th>TLMs sample assertions</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>bus1.socket1.nb_tran_fw(tran.cmd=Rd, tran.tgt=target1, phase=begin_req) [2.005, 1000]</td>
<td>I+II+III</td>
</tr>
<tr>
<td></td>
<td>[2.005, 1000] target1.memory_read()</td>
<td>I+II+III</td>
</tr>
<tr>
<td>B2</td>
<td>bus0.socket0.nb_tran_fw(tran.cmd=Rd, tran.tgt=target0, phase=begin_req) [2.005, 2000]</td>
<td>I+II+III</td>
</tr>
<tr>
<td></td>
<td>[2.005, 2000] target0.memory_read()</td>
<td>I+II+III</td>
</tr>
<tr>
<td>B3</td>
<td>bridge2.socket0.nb_tran_fw(tran.cmd=Wr, tran.tgt=target2, phase=begin_req) [0, 1]</td>
<td>I+II+III</td>
</tr>
<tr>
<td></td>
<td>[0, 1] bus2.socket2 nb_tran_fw(tran.cmd=Wr, tran.tgt=target2, phase=begin_req)</td>
<td>I+II+III</td>
</tr>
<tr>
<td>B4</td>
<td>end_request@init_2 [2.005, 1000]</td>
<td>I+II+III</td>
</tr>
<tr>
<td></td>
<td>[2.005, 1000] bus0.socket2 nb_tran_fw(tran.cmd=Rd, tran.tgt=target0, phase=end_resp)</td>
<td>I+II+III</td>
</tr>
<tr>
<td>B5</td>
<td>bus0.socket2 nb_tran_fw(tran.cmd=Rd, tran.tgt=target0, phase=end_resp) [2.005, 1000]</td>
<td>I+II+III</td>
</tr>
</tbody>
</table>

Evaluation and Analysis of Generated Time Annotated TLM Assertions

In this experiment, the communication primitive functions of all used sockets are instrumented for recording simulation traces. We simulate the design and generate
TLM assertions from the simulation traces using episode mining. The final simulation traces consist of events, TLM communication primitive function calls during simulation, the parameters in each call, and the occurrence time of each function call. The generated trace is then used for episode mining. We show five sample TLM assertions from different sockets in Table 6.5 and also evaluate these assertions according to our proposed standards.

**Assertion B1** asserts that once \( nb\_tran\_fw \) function in bridge1 is called, the \( nb\_tran\_fw \) function in target1 will be called within the range \([2.005\text{ns}, 100\text{ns}]\). Initiator4 calls the \( nb\_tran\_fw \) function in bridge1 to initiate a transaction. The \( nb\_tran\_fw \) function in target1 with phase = end_resp means the end of communication. This assertion thus implies the entire latency of one read transaction communication between initiator4 and target1.

**Assertion B2** asserts that once the initiator0 initiates a read transaction to target0, the memory_read function in target0 will be called. The initiator0 calls the \( nb\_tran\_fw \) function in bridge0 to initiate a transaction. The communication latency is within range \([2.005\text{ns}, 200\text{ns}]\) for different transactions.

**Assertion B3** asserts that the \( nb\_tran\_fw \) call in bridge2 is followed by \( nb\_tran\_fw \) call in bus2 for the write transaction. It also expresses that the delay time of write transaction in the bridge block is less than 1\( \text{ns} \).

**Assertion B4** asserts that once the event end_request occurs in initiator2, the primitive function \( nb\_tran\_fw() \) in bus0 will be called by initiator2. The parameters imply that the transaction is a read transaction, the target is target0 and the phase is begin_resp. The begin_resp is the last phase of the communication between initiator2 and target0. The latency between the event and the function call is within the range \([2\text{ns}, 99.995\text{ns}]\).

**Assertion B5** asserts that the latency between two phases of \( nb\_tran\_fw \) function call by initiator2 is within the range \([2.005\text{ns}, 100\text{ns}]\). The antecedent of B5 indicates the beginning of the communication and the consequent of B5 indicates the ending of that communication.

It can be observed that these assertions always capture the data propagation relations in the interconnection network since we include the function call parameters in simulation traces. Each event or function call represents a high level data operation in the design. For example, the \( nb\_tran\_fw \) means the sending or receiving of a transaction. The mined assertions thus capture the temporal relationships among these high level data operations.

We also conduct experiments on the following two types of simulation traces that
can affect the quantitative time constraint on each assertion.

- Random data trace: Each initiator is allowed to send transaction to any target at random time.

- Concurrent data trace: Multiple initiators send transactions to the same target at approximately the same time.

We observe that the annotated time constraints will change depending on the nature of simulation traces. The value of quantitative real time constraint in concurrent data trace is larger than that in random data trace, which means the latency of data communication is increasing due to the concurrent data requests. Too many such concurrent accesses degrade the performance of the design. Take the assertion B1 as one example. The quantitative real time constraint becomes [2.005, 1500] in the case of concurrent data trace.

Window Constraint’s Impact on TLM Assertion Generation

Window in episode mining is used to prune the search space and thus improve the efficiency of mining algorithm. Episode mining generates candidate assertions among the events within the window. The setting of the window constraint influences the number of generated assertions and also the quality of generated assertions. In this experiment, we show the number of generated assertions for different window constraint in Figure 6.10.

It can be observed that more TLM assertions are generated with the increase of window constraint. Also, the running time the episode mining increases as we enlarge the window constraint, because more events appear in the time window and more candidate episodes needs to be checked by the algorithm. However, if the window is too large, a lot of TLM assertions will be generated. The events/function calls, which belong to different transaction communications, are coincidentally correlated in one assertion. It means the quality of generated TLM assertions are decreasing. In the extreme case, the episode mining is close to the general sequential pattern mining if we set the window constraint to the time length of event sequence.

In our experiment, we set the window constraint as the maximum lifetime of transaction in the simulation trace. In this way, our generated assertions are able to
Figure 6.10: The number of TLM assertions and running time for different window constraints. The experimental design is an AXI based interconnection network. As we increase the window constraint further, the number of generated assertions appears to approach that of sequential pattern mining.

capture the functional behaviors frequently occurring within all transaction communications. Meanwhile, it is less likely to generate the assertions involving events/function calls from two different transaction communications.

6.13 Related Work and Conclusion

The specification mining from software is extensively studied before. Assertion generation though static analysis of source code or a model has been studied in the context of deductive program verification [27]. The deduction of loop invariants can quickly get very complex. Static analysis techniques have been used to learn invariants for assisting software verification. Dynamic analysis techniques like data mining have been used in software to determine system invariants [110]. Daikon [27] runs a software program, observes the value that the program computes and then reports likely invariants that were true over the observed executions. These likely invariants are used in program understanding and evolution. However, the generated likely invariants largely depend on input stimulus of the program and may be spurious with respect to specification. SMArTIC [128] tries to improve the accuracy of mined specification using clustering, filtering, learning and merging techniques.

In hardware assertion generation, most of the work focuses on RTL design [28], [29], [31], [121]. IODINE [31] tries to automatically infer likely invariants by hy-
pothesizing a set of predefined invariant pattern across one or more variables in the
design and then analyzing its dynamic behavior during simulation. The generated
assertions need not be sound and they are usually simple assertions like one-hot en-
coding. Specification patterns mined from correct and erroneous traces can be used
to automatically localize an error [28]. Sequential pattern mining is also applied to
generate assertions from RTL design [121].

The definition of assertions at transaction level and how to efficiently monitor
them in simulation are also explored. A temporal language for SystemC is proposed
in [37], [38], [129]. The hierarchical concept is adopted to construct complex high
layer assertion from simple low layer assertions. The generation of SystemC moni-
tors from given assertions is implemented in Horus [37]. Each monitor is enclosed
in a wrapper observing the channels or events involved in the assertions. However,
the quantitative real time constraint is not considered in the work. The work in [129]
defines TLM assertions in terms of design events or transaction events. The com-
munication action, which is a function call during simulation, is excluded from the
assertion.

The refinement of the assertions at transaction level model to RTL model is dis-
cussed in [126], in which a transactor is specifically designed to map the events/function
calls in transaction level to signals in RTL design. These works do not consider how
to automatically derive the assertions from the transaction level designs.

The performance constraints at system level are also defined and formalized in
[130]. The authors in [130] mainly specified these quantitative constraints in terms
of only events/signals instead of transactions. Moreover, their proposed logic of
constraints are based on post-processing of simulation traces.

Sequential data mining problem was first introduced by Agrawal and Srikant [131].
Many efficient algorithms have been proposed for mining sequential patterns like
GSP [131], SPADA [132], PrefixSpan [133] and SPAM [127]. In SPAM, a depth-
first search strategy is used to generate candidate sequences, and various pruning
mechanisms are implemented to reduce the search space [127]. Mining frequent
episodes from complex sequences has extensive application in financial analysis,
alarm sequence analysis in telecommunication network, and web access pattern
analysis [123].

In hardware, to the best of our knowledge, there have been no prior attempts to
automatically generate system level assertions through data mining.

In conclusion, we have presented an automatic assertion generation from SystemC
designs by mining the simulation traces of the designs. We propose to use a scalable
episode mining algorithm on a transaction level AMBA-based DMA controller and AXI-based interconnection network. We specifically adopt symbolic execution to symbolize the parameters and returned values of function calls to improve the quality of generated assertions. We finally output assertions with annotated quantitative time constraint.
CHAPTER 7

DIAGNOSING ROOT CAUSES OF SYSTEM LEVEL PERFORMANCE VIOLATIONS

7.1 Introduction

One of the main tasks in system level verification is the evaluation of platform performance. Performance specifications are described in the form of latency requirements between two modules or throughput requirement of a single module. Transaction traffic is generated on the input of the platform and the entire platform is simulated. Performance specifications are then verified by checking the generated transaction traces during simulation. When a performance specification is violated, designers have to identify the root cause of the violation in the models using transaction traces. A transaction trace consists of all the operations performed on each transaction during the simulation.

Commercial ESL tools are capable of providing limited statistical analyses of transaction traces [36]. However, the diagnosis process is still ad-hoc and unsystematic. System designers usually have to instrument source code within the models to collect dynamic transaction traces that help localize the root cause. Due to the high speed of system level simulation, an enormous amount of data is generated, with trace files ranging up to several gigabytes. For example, in Huawei’s multi-core TLM platform of a wireless baseband chip, the size of one transaction trace file ranges from 100MB to 7GB. Localizing the root cause of a tough performance violation could take them 2 days to 2 weeks, which tremendously increases the time to market of their product.

In general, the root cause of a performance violation at system level can stem from either hardware resource limitation or inefficiency of software applications in utilizing the underlying hardware resources. In this chapter, we focus on the root causes of the second category.

Determining the root causes of a violating trace requires enormous effort due to the massive size of the trace file. This “finding a needle in a haystack” nature of
system level diagnosis lends itself to a solution using data mining. However, off-the-shelf data mining algorithms are too generic to be applied, or get meaningful answers from, in a specific context. They always generate a huge number of patterns, and it is difficult to use these patterns in diagnosis. Lack of domain context makes the data miner produce arbitrary relationships that might be statistically correlated, but are non-causal. It has been established by GoldMine that providing domain information and knowledge helps to focus the data mining algorithms and filter out random correlation data [29], [46], [48], [134]. GoldMine is an assertion generation tool for RTL. We have extended GoldMine to system level with the same principle in [35].

In this chapter, we present a methodology to extract domain knowledge at the system level to help the data miner isolate behavior relevant to the violation. We also present concurrent mining, an algorithm adapted to identify concurrent patterns in trace data.

Performance violations can usually be traced to concurrent operations on transactions. We identify key *culprit scenarios* through which application software can cause performance violations in the system level design. All our culprit scenarios pertain to concurrent operations on transactions that are potential causes of violation. From our experience with real industrial scenarios, we have identified the following most probable culprit scenarios: (1) multiple modules send transactions to the same target module frequently, and these requests occur at approximately the same time; (2) interleaving read/write access sequences to the same memory target occurs frequently; (3) multiple modules request data from the same memory bank of one target memory at approximately the same time. Frequent occurrences of these concurrent patterns will increase the response time of the corresponding request, and thus lower the performance of the platform.

We provide three types of domain specific information to focus the data miner. The first type of domain knowledge concerns the performance violation time. Any transactions occurring after the time of performance violation need not be analyzed by the data miner, since the root cause of violation is obviously before the violation. This is the first filter we apply to the data traces. The second type of domain knowledge is identifying transaction traces that do not compete for resources with the violating entity. The violating entity can be a transaction for a latency violation and a module for throughput violation. We present a systematic procedure to check for competing transaction traces and retain only these traces. Non-competing transactions are irrelevant when searching for behavior causing latency or through-
put violation. This is the second filter we apply to the data traces. The third kind of domain knowledge we apply is to identify and preprocess the transactions that we want the data miner to analyze for all culprit scenarios. The culprit scenarios involve multiple modules accessing the same target or the same memory bank simultaneously. The data miner needs to be made aware of such information, so it can mine the culprit scenarios we suspect. We analyze the target address in transaction traces and abstract it to the target ID or memory bank ID.

In view of our interest in diagnosing concurrent behavioral patterns that cause violations, we present a concurrent pattern mining algorithm. This algorithm finds concurrent patterns that occur frequently in the transaction traces. The discovered patterns are in terms of transaction operations in the transaction traces. We require that the operations in the patterns occur at approximately the same time, because the performance violation is an accumulative effect of many transactions in a period of time. In order to find these patterns, we use an interval window sliding with time. Transaction operations that occur within the interval window are considered as having occurred at the same time. Finally, these frequent concurrent patterns are reported as the most likely root causes of performance violations. Existing frequent itemset mining [3] algorithms will not be able to produce the result we need, since they do not restrict themselves to concurrent behaviors and cannot be applied in diagnosing performance violations.

We provide the three culprit scenarios to the concurrent pattern algorithm, by preprocessing and augmenting the transaction trace data to include information relevant to the corresponding culprit scenario. As an example, for the interleaving read/write culprit scenario, we remove all other attributes but retain the read/write attribute, the source ID attribute, the target/memory bank ID attribute, and the operation occurrence time in the transaction traces. Only the transaction traces of these attributes are necessary to characterize this culprit scenarios. Consequently, the root causes output by the concurrent pattern mining will belong to this culprit scenario category.

When a human being debugs and diagnoses such a violation, he/she implicitly applies certain filters based on knowledge of context and relevance. The difficult part of the process for a human is sifting through and analyzing large amounts of data. We systematize human intuition as generic domain knowledge, and culprit scenarios use the data miner simply to hasten the process of searching through large data. In essence, we present a flow for manipulating our core data mining algorithm to provide meaningful answers to our problem. We do not use the data mining as a
single point solution, since our and others’ experience shows that the value of the 
mined information is highly dependent on context [28], [29], [121]. We use domain 
knowledge to remove irrelevant traces as well as to preprocess and augment the 
transactions with the behavior we want the miner to focus on. We also provide 
three bins for the miner to classify its results into, using the culprit scenarios.

We have implemented a complex, realistic experimental platform similar to an 
industrial platform to facilitate industrial collaboration. The platform uses hier-
archical AXI buses to connect a cluster of CPUs, a cluster of DSPs, two DMA 
controllers, and two eight bank memories. CPUs and DSPs serve as initiators, and 
DMA controllers and memories serve as targets during communication. Since there 
is no publicly available complex TLM platform for academic/research use, we also 
release this platform in the public domain [124]. We evaluate our technique on this 
platform. Our results show that the domain knowledge can reduce the number of 
transaction traces by up to $92.8\%$. Also, the concurrent pattern mining pinpoints the 
root cause of one violation to one of fewer than 10 patterns among 100000 transac-
tion traces. Without domain knowledge, more than 900 patterns are generated for 
each culprit scenario.

7.2 Preliminaries

In this section, we provide the background about transaction traces and performance 
violation. We also give several definitions used in the mining algorithm.

7.2.1 Transaction Traces

In TLMs, the communication between computation modules is done through channels. 
These channels provide a set of standard communication primitives in order 
to hide low level protocol details. The data and protocol related attributes, such as 
address and data length, are encapsulated into transactions [55]. The TLMs serve 
as the executable virtual platform of the entire system.

A transaction is structural data transmitted between modules. Each transaction 
has multiple protocol related attributes, such as command type, address, and data 
length. Each transaction has a lifecycle, from its creation to its release. During 
its lifecycle, modules in the design perform operations on the transaction. Each
transaction is assigned a unique ID to track all operations on the transaction during its lifecycle.

During a simulation, we record the transaction traces into a database. In addition to the transaction attributes and ID, we also record the operation attributes. The operation attributes include operation name, operation module, and operation time. Examples of operation name are “forward requesting”, “pushing into some fifo”, “popping out from some fifo” etc.. Operation module specifies which module performs the operation on the transaction. We also annotate each operation with a time property to record when the operation occurs. The structure of an operation in the transaction traces is a tuple, shown as follows:

< ID, Attr₁, Attr₂, ..., Attrₙ, Op name, Op mod > @time

A sample transaction trace during simulation is shown below. Operations in the transaction traces are sorted by the annotated time property. From the sample shown below, we can see that all operations on transaction 1 during its lifecycle are recorded from the creation operation to the release operation. Also one transaction operation may occur multiple times in the transaction trace.

< 1, read, 0x1234, ...16, create, initiator1 > @200ns,
< 1, read, 0x1234, ...16, fw_req, initiator1 > @200ns,
< 2, write, 0x1278, ...32, fifo_push, moduleA > @207ns,
...
< 1, read, 0x1234, ...16, bw_resp, target1 > @500ns.
< 1, read, 0x1234, ...16, release, target1 > @500ns,
...

An occurrence of an operation in the transaction trace is also called an event. Therefore, we call a transaction trace an event sequence. In this chapter, a pattern is a collection of events from the event sequence.

7.2.2 Performance Violations in TLMs

Performance evaluation is one of the main tasks in transaction level modeling of systems. Performance specifications include latency specification and throughput specification. The specifications impose constraints on communication latency and a module’s processing capability of transactions.
In TLMs, one-time communication is initiated by the \textit{initiator} module and responded by the \textit{target} module.

\textit{Latency specification} refers to the allowed time limit of transmitting a transaction between an initiator and a target in the model. The transmission of a transaction corresponds to one operation in the initiator and one operation in the target. We require that all transactions transmitted between a specified initiator and target satisfy the latency specification.

\textit{Throughput specification} refers to the minimum total number of bytes of transactions processed by a module within a time unit. The processing of a transaction corresponds to one operation of the transaction. For example, we can measure the throughput of a module by accumulating all transactions received by this module within a time unit.

We claim that the performance specification in the model is \textit{violated} when at least one transaction latency exceeds its corresponding specification or at least one module’s throughput is smaller than its corresponding specification.

Given a performance specification and transaction traces, if there is a performance violation in the transaction traces, our problem is how to discover the frequent concurrent patterns that cause this violation.

\subsection*{7.2.3 Concurrent Patterns}

Our approach tries to mine concurrent patterns from an event sequence as the root causes of performance violations. Concurrent pattern is similar to parallel episode defined in [123]. It is different from the episode definition of Chapter 6 in that it does not impose an ordering constraint on the events in the generated patterns. We formally define concurrent patterns in this section.

Let $E$ be a set of distinct events. An event occurrence is denoted by $e@t$, where $e \in E$ and $t > 0$. $t$ denotes the time of occurrence of event $e$. Let $S$ be an ordered list of event occurrences. Let us denote $S$ as $<e_1@t_1, e_2@t_2, \ldots, e_n@t_n>$ where $\forall i \ e_i \in E$ and $t_1 \leq t_2 \ldots \leq t_n$. Note that one event $e$ may occur multiple times in $S$. We call $S$ the \textit{event sequence}. In Figure 7.1, we show an example of sequence database.

A \textit{concurrent pattern} is defined as a set of multiple events that occur within a given \textit{interval} in an event sequence. We are interested in concurrent patterns that include at least two events. It should be noted that a concurrent pattern does not
impose ordering constraints on the events. In the event sequence example shown in Figure 7.1, if the given interval is 3, \( \{e_1, e_2\} \), \( \{e_2, e_4\} \), and \( \{e_2, e_5\} \) are all sample concurrent patterns. To discover the concurrent pattern, we slide the interval window on the time axis.

The same concurrent pattern may occur multiple times in an event sequence. In Figure 7.1, pattern \( \{e_1, e_2\} \) occurs four times. If the concurrent pattern occurs in one interval window, then it is counted as one occurrence. The occurrence frequency of a concurrent pattern is called the support of the pattern in the event sequence. Given a minimum support threshold, the concurrent pattern is frequent if its occurrence frequency is greater than the threshold.

We require that the events in a concurrent pattern occur within a given interval rather than at exactly the same time. Within the context of our application, two events that occur closely enough may lead to a performance violation. Since performance violation is a cumulative effect of many transaction operations during simulation, we require that the concurrent patterns occur frequently.

7.3 Concurrent Pattern Mining Algorithm

Our concurrent pattern mining algorithm generates frequent concurrent patterns from an event sequence. The algorithm is shown in Algorithm 2. The top algorithm flow is the same as that defined in Algorithm 1. We elaborate each function called in the algorithm.

In the mining algorithm, \( E \) is the set of distinct events, and \( E_{\text{Seq}} \) is the event sequence. \( \text{Min}\_\text{supp} \) is the minimum support threshold, and \( \text{Interval} \) is the interval value for concurrent pattern mining. \( \text{ConPat} \) is the set of all discovered concurrent patterns. \( L_i \) is the set of frequent patterns having \( i \) events. The algorithm derives candidate patterns with \( i + 1 \) events from the frequent patterns with \( i \) events, which is based on the Apriori property [3]. All nonempty subsets of a frequent concurrent
Algorithm 2 Concurrent Pattern Mining algorithm

ConPatMine($E, E_{Seq}, Min_{supp}, Interval$)

1: $ConPat = \emptyset$
2: $L_1 = Con_{Check}(E, E_{Seq}, Min_{supp}, Interval)$
3: for ($i = 1; L_i \neq \emptyset; i++$) do
4: $C_{i+1} = Cand_{Gen}(L_i)$
5: $L_{i+1} = Con_{Check}(C_{i+1}, E_{Seq}, Min_{supp}, Interval)$
6: $ConPat = ConPat \cup L_{i+1}$
7: end for
8: return $ConPat$

Algorithm 3 Candidate Generation algorithm

Cand_{Gen}(L_i)

1: $C_{i+1} = \emptyset$
2: for all $P_m, P_n \in L_i$ do
3: $P_{mn}^e = P_m^e \cap P_n^e$
4: if $|P_{mn}^e| == (i - 1)$ then
5: $C_{i+1} = C_{i+1} \cup \{P_m^e \cup P_n^e\}$
6: end if
7: end for
8: return $C_{i+1}$

Algorithm 3 is the candidate pattern generation algorithm. In iteration $i$, for all two candidate pattern pairs ($P_m, P_n$) from $L_i$, $Cand_{Gen}$ tries to join them together to form a new pattern $P_{mn}$ with $i + 1$ events. We require that there are $i - 1$ common events in $P_m$ and $P_n$ when joining them, which is shown in line 4. $P_m^e$ represents the set of all events in $P_m$. We also show the generation process in Figure 7.2. $Cand_{Gen}$ is allowed to join $\{e_1, e_2\}$ and $\{e_1, e_3\}$ to form $\{e_1, e_2, e_3\}$. It is not allowed to join $\{e_1, e_2\}$ and $\{e_3, e_4\}$ to form $\{e_1, e_2, e_3, e_4\}$.

Not all candidate patterns are frequent concurrent patterns. The function $Con_{Check}$ slides the interval window along the time axis to compute the frequency of each concurrent pattern in $C$, and returns the set of frequent concurrent patterns. The detailed algorithm is shown in Algorithm 4. Lines 3 – 16 check the frequency of pattern $P$ in the event sequence. $E_{Seq}[i].e$ represents the $i_{th}$ event in event sequence $E_{Seq}$, and $E_{Seq}[i].t$ represents the occurrence time of the $i_{th}$ event. $j$
Figure 7.2: Candidate pattern generation. $C_i$ is generated from $L_{i-1}$. $L_i$ is the subset of frequent patterns in $C_i$.

is the first event in $E_{Seq}$ after the current window (line 6). The current interval window is $[E_{Seq}[i].t, E_{Seq}[i].t + \text{Interval})$. If the first event in the current window is not in $P^e$ (line 5), the window is slid to next event (line 14). Otherwise, it computes all events in the current window (line 7). If all events in $P^e$ appear in the current window (line 8), it increases the frequency of $P$ by one (line 9) and slides the window to the $j^{th}$ event (line 9). If not all events in $P^e$ appear in the current window, the window is slid to the next event (line 11). Finally, if the occurrence frequency is larger than the threshold support, the candidate pattern is considered as a frequent pattern (line 17).

7.4 Mining Concurrent Patterns for Root Cause Localization

In this section, we first introduce transaction trace management using a database. We then explain the entire framework of our approach to discover the concurrent patterns in the transaction traces when a performance violation occurs.

7.4.1 Transaction Trace Management

Our approach is based on an off-line analysis of generated transaction traces. We use a SQL database to manage the transaction traces. The trace database facilitates efficient performance specification checking and concurrent pattern mining.

Figure 7.3 shows our procedure for managing the transaction traces using the SQL database. We extend the SystemC TLM library [55] for recording the transaction traces. Each operation on a transaction in SystemC TLM is a call to a primitive function provided in the library. For example, the initiator calls the $\text{nb}_\text{transport}_\text{fw}$ function in the TLM library to send a transaction to a target module. For every prim-
Algorithm 4 Candidate Check algorithm

\textbf{Con\_Check}(C, E\_Seq, Min\_supp, Interval)

\begin{enumerate}
\item $L = \emptyset$
\item for all $P \in C$ do
\item \hspace{1em} $freq = 0$; $i = 0$
\item \hspace{2em} while $E\_Seq[i].t < E\_Seq[\text{MAX}].t$ do
\item \hspace{3em} if $E\_Seq[i].e \in P^e$ then
\item \hspace{4em} $j = \min_j (E\_Seq[j].t > (E\_Seq[i].t + \text{Interval}));$
\item \hspace{4em} $\text{Event\_in\_Win} = \bigcup_{i \leq k < j} \{E\_Seq[k].e\};$
\item \hspace{4em} if $P^e \subseteq \text{Event\_in\_Win}$ then
\item \hspace{5em} $freq + +$; $i = j$
\item \hspace{4em} else
\item \hspace{5em} $i + +$
\item \hspace{4em} end if
\item \hspace{3em} else
\item \hspace{4em} $i + +$
\item \hspace{3em} end if
\item \hspace{2em} end while
\item \hspace{1em} if $(freq \geq \text{Min\_supp})$ then
\item \hspace{2em} $L = L \cup P$
\item \hspace{1em} end if
\item \hspace{1em} end for
\item \hspace{1em} return $L$
\end{enumerate}

---

**Figure 7.3:** Transaction trace management using SQL database.

itive function, our extension implements a callback function to write the operation with its time, transaction, and parameters to the SQL database using the database programming interface [135].

Our transaction trace management using the SQL database is much more efficient than that provided by commercial ESL tools [36]. Users are allowed to select the modules for which they wish to record traces. We also use a standard SQL database tool [136], which provides highly efficient and optimized data management of large transaction traces. To facilitate the performance violation checking and concurrent pattern mining, we provide API interfaces to access the trace database.
7.4.2 Framework of Our Approach

Figure 7.4 shows the complete flow of our method. Given a transaction trace database and latency/throughput specification, we conduct an off-line analysis on the trace database to check whether there is any latency/throughput violation. The identified violation information, which includes the violation time and violated transaction, provides domain knowledge I&II for filtering the irrelevant transaction traces. The address space of each memory or bank provides domain knowledge III for target/memory bank ID abstraction.

After preprocessing the transaction traces, we remove unnecessary attributes of the transaction according to the category of culprit scenarios. The remaining transaction traces are mapped to event sequences for concurrent pattern mining. We target the three main categories of culprit scenarios as shown in Figure 7.4. Culprit scenario I represents concurrent request scenario; Culprit scenario II represents interleaving read/write scenario; and Culprit III represents bank conflict scenario. According to the three different scenarios, we route the concurrent pattern mining algorithm using the relevant transaction attributes. The generated concurrent patterns pinpoint the root cause pertaining to the corresponding culprit scenario. The output patterns are listed by mining algorithm according to their support value in the event sequence. The most frequently occurring patterns in every culprit scenario are the most likely root causes of the performance violation.

7.4.3 Checking Performance Violations

Using the API interfaces we have implemented, it is easy to check the latency specification and throughput specification. For latency specification, we first specify the operations in the source and target as query conditions to get all related transactions, and then compute each transaction latency with the corresponding operation time. For example, if we want to check the transaction latency between modules A and B, the SQL query in the corresponding API is as follows:

\[
SELECT T.ID, T.rec_time from T where T.src = A and T.dst = B and (T.rec_time - T.send_time > latency_spec).
\]

As shown in Figure 7.4, if there is no performance specification violation, the flow exits. Otherwise, the violation time and the violated transaction ID (latency violation)/Module ID (throughput violation) are returned from the query. Violation time is used as domain knowledge I, and violated transaction/Module ID is used as
Figure 7.4: The flow for root cause localization of performance violation using data mining. The discovered root causes are in the form of generated concurrent patterns.

domain knowledge II in the next step.

7.4.4 Preprocessing Transaction Traces with Domain Knowledge

The preprocessing step uses the domain knowledge to prepare the transaction traces for concurrent pattern mining.

Filtering Irrelevant Transaction Traces
Algorithm 5 Detect_Competition_by_DK_II algorithm

\[\text{Trace\_Filter\_DK\_II(Vio\_Tran\_ID, Trace\_DB)}\]

1:  \(M\_Queue = \text{Select(Trace\_DB, Vio\_Tran\_ID)}\);
2:  \(\text{Tran\_Set} = \{\text{Vio\_Tran\_ID}\};\)
3:  \(\text{while } M\_Queue \neq \emptyset \text{ do}\)
4:    \(\text{module} = M\_Queue\_\text{pop\_front}();\)
5:    \(\text{if } (\text{Is\_processed(module)} == \text{False}) \text{ then}\)
6:      \(\text{Tran} = \text{Select(Trace\_DB, module)};\)
7:      \(\text{Tran\_Set} = \text{Tran\_Set} \cup \text{Tran};\)
8:      \(\text{Is\_processed(module)} = \text{true};\)
9:      \(\text{for all } m\_inc \in \text{Select(Trace\_DB, Tran)} \text{ do}\)
10:     \(M\_Queue\_\text{push\_back}(m\_inc);\)
11:    \(\text{end for}\)
12:  \(\text{end if}\)
13:  \(\text{end while}\)
14:  \(\text{return Tran\_Set};\)

\textbf{Domain knowledge I:} Domain knowledge I concerns the time when a performance violation occurs. We use this violation time to filter the irrelevant transaction traces. Only transaction traces occurring before or at the time point of violation are used for further mining. Transaction traces occurring after the violation time cannot cause the violation.

\textbf{Domain knowledge II:} Domain knowledge II is the violated transaction/module ID when a performance violation occurs. We use this violated transaction/Module ID to filter the irrelevant transaction traces. Only the traces of transactions that compete for resources with the violated transaction or module are used for further mining.

We describe our procedure for finding the traces of transactions competing for resources with the violated transaction. We first get the entire violated trace of the transaction by using the API interfaces. We then identify all modules appearing in the trace. Any transaction having operations occurring in these modules is a relevant transaction. In addition, the transaction records with operations that occur in some modules before these modules in the transaction traces are also necessary.

The formal algorithm for filtering irrelevant transaction traces with violated transaction ID is shown in Algorithm 5. The \textit{Select} function in line 1 gets all modules in the violated transaction trace. For each unprocessed module (line 5), any transaction having operations in that module is retained (lines 6 and 7). Recursively, for each retained transaction, the algorithm identifies the modules in that transaction trace (lines 9 and 10). Finally, for the violated transaction, all relevant transaction
traces are extracted, and the irrelevant ones are filtered.

Figure 7.5: Figure showing how domain knowledge II is used to filter irrelevant transaction traces. The red arrow trace from initiator 2 to target 1 shows a latency violated transaction trace. Some operations in the transaction trace are irrelevant to the performance violation.

In the example shown in Figure 7.5, the identified modules in the violated transaction trace are *initiator 2*, *B*, *C*, *D*, and *target 1*. The traces of any transaction record having operations occurring in those modules are kept for mining. If a transaction is sent from module *initiator 1* to module *C*, then the corresponding transaction traces are also kept for further mining.

In case the throughput specification of a module is violated, the algorithm is similar to that in Algorithm 5. Given a violated module ID, we get all transactions having operations in this module. Recursively, for each relevant transaction, the algorithm identifies the modules in that transaction trace. In Figure 7.5, let us assume that the throughput specification of module *C* is violated. The transaction traces in *initiator 1*, *initiator 2*, *A*, and *B* are included for mining if there are transactions sent from *initiator 1* and *initiator 2* to module *C*.

Target/Bank Abstraction

*Domain knowledge III*: Domain knowledge III is the address space information of each target module. We use it to abstract the target ID or memory bank number from the concrete address in the transaction traces. Once we have gotten the target ID information for a transaction operation, we can determine whether two transactions are sent to the same target. With target memory bank information, we can determine whether two transactions are sent to the same memory bank.

The target ID and bank number information is not provided in the transaction attributes, and the mining engine is unaware of such information. Sending a transaction to a single concrete address does not lead to a performance violation. How-
ever, sending a group of transactions to the same target may result in a performance violation. Therefore, this target ID or memory bank information helps the mining engine to produce more meaningful concurrent patterns that are more relevant to the performance violation.

As shown in Figure 7.4, our flow adds the symbolic target memory or bank ID attribute to the transaction traces. The value of this attribute is determined by its target address of the transaction and the address space information. We check which target memory or bank the target address belongs to. After applying domain knowledge III, we obtain the relevant transaction traces with symbolic target memory or bank IDs.

### 7.4.5 Mining Concurrent Patterns from Three Culprit Scenarios

The performance violation is always an accumulative effect of many concurrent patterns rather than just one. Therefore, we require the concurrent patterns to be frequent in the transaction traces. We provide three culprit scenarios to the concurrent pattern mining algorithm by including only the information relevant to the corresponding culprit scenario. As a result, the root causes output by the concurrent data mining engine will belong to this culprit scenario category.

**Concurrent Request Patterns**

A concurrent request pattern refers to multiple transactions from initiators being sent to the same target within a given interval. We use the interval value to express such approximate concurrency. Multiple concurrent accesses to the same target may result in a very high response time. Modules routing access to the target are thus at high risk of becoming congested. Therefore, concurrent request patterns are usually one of the root causes of performance violations.

In the concurrent request culprit scenario, we remove all attributes in the transaction traces but the source module ID, the target module ID, and the operation time. We then map the transaction traces into an event sequence. As shown in Figure 7.4, the event $e$ is a pair containing the source module ID and the target module ID. We calculate the target module ID in the target/bank abstraction step of Figure 7.4. The time $t$ is the same as the operation time in the transaction traces.
Interleaving Read/Write Patterns

Interleaving read/write pattern refers to a read/write or write/read sequence that is issued to the same target memory within a given interval. The memory controller module takes several cycles to reverse the direction of the memory data bus when read and write operations are interleaved. Thus, it gives tremendously different response times to different input request patterns [137]. For example, issuing multiple consecutive reads tends to be much faster than issuing the interleaving read and write patterns. Therefore, we consider the concurrent interleaving read/write patterns one of the root causes of performance violation.

In the interleaving read/write culprit scenario, we remove all attributes in the transaction traces but the source module ID, the target memory ID, the Rd/Wr and the operation time. We then map the transaction traces into an event sequence. As shown in Figure 7.4, the event $e$ is a triple containing the operation module name, the read/write attribute and the target module. We calculate the target module in the target/bank abstraction step of Figure 7.4. The time $t$ is the same as the operation time in the transaction traces.

Bank Conflict Patterns

Bank conflict pattern refers to several memory accesses being issued to the same memory bank in a target within a given interval. This pattern will lead to a bank conflict in the memory, which results in a longer response time for memory access, since the memory controller has to serialize the accesses to the same memory bank. However, the memory can respond quickly to the access of different banks in the memory. Therefore, we consider the bank conflict patterns as one of the root causes of performance violation.

In the bank conflict culprit scenario, we remove all attributes in the transaction traces but the source module ID, the target module ID, the target bank ID and the operation time. We then map the transaction traces into an event sequence. As shown in Figure 7.4, the event $e$ is a triple of operation module ID, memory bank ID and target ID. We calculate the target memory bank and the target ID in the target/bank abstraction step of Figure 7.4. The time $t$ is the same as the operation time in the transaction traces.
7.5 A Case Study

We reuse the platform in Figure 6.7 from Chapter 6 as a case study in this section.

7.5.1 Relating Concurrent Patterns to Performance Violations

In the first experiment, we intentionally generate input transactions in five initiators according to our three categories of concurrent patterns. We demonstrate the relation between concurrent patterns and performance violations.

Figure 7.6: Figure showing the relations between concurrent patterns and performance violations. The x-z plane plots the transaction latency versus time, while the x-y plane depicts the occurrences of different patterns at different times, where each frequent occurrence is arranged along the y-axis. Concurrent requests, interleaving read/write accesses, and bank conflict accesses are depicted as color coded triangles or trapezoid.

Figure 7.6 shows the observed transaction latency together with the occurred patterns in transaction traces. In the figure, the y-axis represents different concurrent patterns. \((I_i, T_j)\) means that initiator \(i\) sends transactions to target \(j\). \((I_i, R, T_j)\) means initiator \(i\) sends read transactions to target \(j\). \((I_i, B_k, T_j)\) means that initiator \(i\) sends transactions to bank \(k\) of target (memory) \(j\).

It can be observed that the transaction latency will increase tremendously if we have concurrent requests, interleaving read/write accesses, and memory bank conflict accesses in transaction traces. As an example, in the culprit scenario II shown in the middle of the figure, the latency specification is that the transaction latency between CPUs and memories should be less than 100ps.

In the initiators, CPU0 (initiator 0) keeps sending read transaction to Memory 2 (target 3), and DSP0 (initiator 3) keeps sending write transaction to the same target memory. We control the sending time difference in CPU0 and DSP0 within 10ps. The sending frequency is one transaction per 50ps.
We observe that the latency of transaction between CPU0 (initiator 0) and Memory 2 (target 3) becomes 1200ps, which violates the specified latency specification. However, if DSP0 (initiator 3) does not send write transactions to Memory 2 (target 3) in this case, the latency is only 7.5ps. The interleaving read/write pattern increases the response time of Memory 2 (target 3).

7.5.2 Domain Knowledge for Filtering Irrelevant Traces

In this experiment, we evaluate the effectiveness of applying domain knowledge I&II for filtering irrelevant transaction traces. We run three test cases corresponding to three categories of concurrent patterns and generate 5000 transactions in each case. We then record the transaction traces during simulation. We analyze the transaction traces and calculate how many transactions are retained after applying each kind of domain knowledge. There might be multiple latency/throughput violations. We use the earliest latency violation in this experiment.

Table 7.1: Applying domain knowledge I and II to filter the irrelevant transaction traces for mining. The table entries show the retained number of transactions after preprocessing of the transaction traces.

<table>
<thead>
<tr>
<th>Testcases</th>
<th>Total number of transactions</th>
<th>Apply domain knowledge I</th>
<th>Apply domain knowledge II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case I</td>
<td>5000</td>
<td>1125</td>
<td>396</td>
</tr>
<tr>
<td>Case II</td>
<td>5000</td>
<td>1030</td>
<td>412</td>
</tr>
<tr>
<td>Case III</td>
<td>5000</td>
<td>2130</td>
<td>1801</td>
</tr>
</tbody>
</table>

The latency specification is that the transaction latency between CPUs and DMACs should be less than 250ps. In Case I, three CPUs send transactions to the target DMAC0 at the same time and all other DSPs randomly generate transactions, and the violation occurs at 1005ps. The entire simulation runs for 5000ps. In Case II, CPU0 sends read transactions to Memory 1 (target 3) and CPU1 sends write transaction to Memory 1 (target 3) simultaneously, and CPU2 and other DSPs randomly generate transactions. The earliest violation occurs at 1200ps. In Case III, all three CPUs send the transactions to bank 4 in Memory 0 (target 2), and the concrete address within bank is randomized. All other DSPs randomly generate transactions. The earliest violation occurs at 2700ps.

From Table 7.1, it can be observed that domain knowledge I&II significantly reduces the number of transaction traces for further mining. Taking Case I as an
example, domain knowledge I can reduce the number of transactions by 77.5% and
domain knowledge II can reduce the number of transactions by 64.8%. In Case III,
domain knowledge II can be used to filter out the random transaction traces from
DSPs to DMA controller. There are about 15% irrelevant transaction traces.

7.5.3 Domain Knowledge for Improving Mining Results

In this experiment, we randomly generate input transactions in initiators and then
simulate our platform. We demonstrate that the domain knowledge helps improve
mining results. We apply our approach with and without domain knowledge I and
II. Without domain knowledge III, the mining engine is not capable of producing
the three categories of patterns.

Figure 7.7: Figure showing the number of generated concurrent patterns with and
without domain knowledge. The domain knowledge reduces the number of
discovered concurrent patterns to less than 10.

Figure 7.7 shows the number of generated concurrent patterns with and without
domain knowledge. It can be observed that the number of generated patterns is
reduced from 1000 to less than 10. The reason for this is that multiple irrelevant
operations occur within the given interval along with the relevant operations in the
transaction traces. As a result, several irrelevant patterns are generated and the
number of patterns generated suffers from combinatorial explosion. However, with
domain knowledge, these irrelevant operations are removed before mining and the
mining engine is able to discover more accurate and fewer patterns as the true root
causes. As seen in Figure 7.7, the mining engine discovers less than 10 patterns for
each category as root causes.
7.5.4 Sample Concurrent Patterns Analysis

Table 7.2: Sample concurrent patterns discovered using concurrent pattern mining. Ix represents initiator x. Tx represents target x. Bx represents bank x. W represents write operation. R represents read operation. Therefore, (I1, B2, T2) means initiator 1 sends request to bank 2 of target 2.

<table>
<thead>
<tr>
<th>ID</th>
<th>Concurrent Pattern</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(I1, T3)-(I3, T3)</td>
<td>Concurrent request</td>
</tr>
<tr>
<td>2</td>
<td>(I0, T3)-(I4, T3)-(I3, T3)</td>
<td>Concurrent request</td>
</tr>
<tr>
<td>3</td>
<td>(I0, W, T3)-(I1, R, T3)</td>
<td>Interleaving read/write</td>
</tr>
<tr>
<td>4</td>
<td>(I0, R, T3)-(I1, W, T3)-(I2, R, T3)</td>
<td>Interleaving read/write</td>
</tr>
<tr>
<td>5</td>
<td>(I1, B5, T3)-(I1, B5, T3)</td>
<td>Bank conflict</td>
</tr>
<tr>
<td>6</td>
<td>(I1, B2, T2)-(I1, B2, T2)-(I2, B2, T2)</td>
<td>Bank conflict</td>
</tr>
</tbody>
</table>

The memory model is pipelined. One transaction does not need to wait for the previous transaction to finish unless there is a bank conflict. A bank conflict arises when one transaction is trying to access a bank that is currently being accessed by a previous transaction. In such a case, this transaction is pushed into a pending queue in the memory model. Once the previous transaction has finished and released the bank, this transaction is popped out of the pending queue and is processed. The waiting time in the queue contributes to a large transaction latency. In Table 7.2, for pattern 5 and 6, B5 in T3 and B2 in T2 are repeatedly accessed, which leads to severe bank conflicts, a long pending queue and large latency. The throughput of the memory target is also violated.

Read/write patterns are another causes for latency. A read followed by a write or vice versa will require the reversal of the direction of the data bus and thus result in extra delay. Pattern 3 and 4 represent interleaving read/write patterns. Pattern 3 is a simple read after write while pattern 4 takes one write in between two reads.

Patterns 1 and 2 represent multiple initiators requesting T3 concurrently. Because of the pipelined memory model, these requests can be accepted without extra delay. The reason that concurrent requests are correlated to latency violation is the higher probability of having interleaving read/write and bank conflicts in multiple requests.

7.5.5 Experiments with Different Threshold Value and Interval Size

The interval value and support threshold value have a large impact on the discovered concurrent patterns. Choosing a threshold value is a trial-and-error process. We
rank all generated patterns by their absolute support value in transaction traces. If we are not able to generate correct concurrent patterns as root causes, we can decrease the threshold value in order to generate more concurrent patterns.

The interval value depends on the transaction latency in the system. If the interval is too large, the discovered concurrent patterns may not lead to performance violation. If the interval is too small, we may miss some important concurrent patterns, which are the true root causes of performance violations. Figure 7.8 shows the number of generated concurrent patterns as we increase the size of the interval constraint. It can be observed that the number of generated patterns increases as we enlarge the size of interval. When we increase the interval size, many events are coincidentally correlated in one pattern. As a result, the discovered concurrent patterns might not be the real root cause of violation. Also it can be observed that few concurrent patterns are discovered when the interval size is less than 100\(\text{ns}\). Changing the interval value reduces the number of patterns. However, it does not reduce the number of transactions in the traces since the window is slid on the time axis of all transaction traces.

In Figure 7.8, we find that all injected concurrent patterns are discovered when the interval is set as 100\(\text{ns}\), which is the largest transaction latency in this experiment. Intuitively, the interval gives the time range in which transactions can compete for resources with each other. Therefore, we set interval value as the average transaction latency before the occurred violation in transaction traces in practice. The running time of mining algorithm heavily depends on the number of events in the trace. In our experimental platform, the number of events is less than 100 and the running time is less than one second.

![Figure 7.8: The number of generated concurrent patterns as we increase the size of interval constraint in concurrent pattern mining.](image)
7.6 Related Work

Many previous studies about system level TLM verification focus on functionality instead of performance [138]. Assertion based verification is also employed in verifying system level TLMs [37], [139].

Performance constraints at the system level are defined and formalized in the form of logic of constraints [130]. However, they do not solve the problem of root cause localization when performance constraints are violated; also their method is not based on standard TLM. Online monitoring of the simulation is impractical since we need to store the transactions in memory and dynamically check the assertion violation. This process not only consumes a lot of memory, but also slows down the simulation speed. Moreover, the root cause localization for a performance violation has not been considered.

Data mining techniques have been explored for hardware verification [28], [29], [46], [121]. The work in [140] uses a hardware logging mechanism and a data-mining approach to automatically report abnormal instruction timings and the context of occurrence of these instructions. The mining algorithm is based on frequent itemset mining, and also the model is not a standard TLM. Moreover, their method tries to find the frequent contention patterns instead of diagnosing performance violations. In [141], frequent itemset mining is also applied for TLM functional verification. However, it simply counts the number of different transactions and is not related to performance diagnosis.

Concurrent pattern mining in this chapter is inspired from multiple pattern mining algorithms such as the Apriori algorithm [3], sequential pattern mining [3], and episode mining [123]. Our concurrent pattern mining is similar to parallel episode mining. However, episode mining always assumes discrete time points [123]. Episode mining slides the interval window at a step of unit time and calculate the number of windows the pattern appears in. The support value of a pattern is the number of windows, in which the pattern appears, divided by total number of windows. In our context, we do not limit the transaction occurrence time to discrete time points. We calculate the absolute support of patterns and also the window can slide by an interval size each time when one occurrence of a pattern is found.
7.7 Summary

In conclusion, we have presented a methodology to localize root causes of performance violations in TLMs. The discovered root causes are presented in the form of concurrent patterns mined from transaction traces. The methodology can also easily be extended to other new culprit scenarios related to performance violation.
CHAPTER 8
CONCLUSION

We have presented a suite of techniques that are a significant departure from traditional formal or simulation based verification approaches. The presented algorithms are based on static/dynamic analysis of design code and data mining from dynamic simulation traces. The techniques presented here can blend seamlessly into current chip design cycle.

We use hybrid analysis of RTL source code for systematic and scalable input vector generation for simulation based verification. Static analysis of RTL source code for test generation is not scalable to large designs due to the space explosion, and the dynamic simulation only captures partial design behaviors. The hybrid analysis offsets both disadvantages and improves the scalability of input vector generation for practical design.

Data mining opens the door to many difficult problems in hardware verification as the hardware system is becoming increasingly complex. In GoldMine, data mining algorithm is successfully applied to learn the invariant rules as RTL assertions from dynamic simulation traces. We have proposed a methodology for attaining coverage closure of design validation using GoldMine spurious assertions. Our algorithm always converges and captures the complete functionality of each output of a design on convergence. It always results in a monotonic increase in simulation coverage and finally attains coverage closure with respect to the input space coverage.

Simply applying data mining without any guidance will generate meaningless results. In other words, the value of the mined information (knowledge) depends heavily on application context. Our word level assertion generation technique could improve the readability and expressiveness of mined assertions by providing word level features to guide the data mining algorithm. These word level features are discovered from RTL source code.

ESL verification is indispensable for complex SoC designs. We lift the automatic assertion generation methodology from RTL to ESL. We have attempted to use sequential pattern mining and episode mining for system level assertion gener-
ation from TLMs. We have demonstrated that episode mining is more scalable in our context, and also generates a more compact set of assertions. The TLM assertions generated by the episode mining algorithm have higher quality in terms of the evaluation standards defined by us.

We have presented a concurrent pattern mining technique for troubleshooting performance violations in ESL verification. The root causes of performance violations are attributed to concurrent and frequent accesses of shard resources, which are mined from transaction traces. We systematize human intuition as generic domain knowledge, and culprit scenarios use the data miner simply to hasten the process of searching through large data.
REFERENCES


160


