國立臺灣大學電機資訊學院電子工程學研究所

碩士論文

Graduate Institute of Electronics Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

# 利用約束測試模式和斷言驗證進行 RTL Bug 自動定位

# Automatic RTL Bug Localization by Constrained Pattern Generation and Assertion Validation

陳孟宏

Meng-Hung Chen

指導教授: 黃鐘揚 博士

Advisor: Chung-Yang (Ric) Huang Ph.D.

中華民國 113 年 8 月

August, 2024

doi:10.6342/NTU202403664

# Acknowledgements

在這兩年的碩士旅途中，我有幸獲得許多人的幫助，在此要一一表達由衷的感謝。

首先，我由衷感謝我的指導教授，黃鐘揚（Ric）教授，謝謝教授的博學多聞和豐富的業界經驗，讓我在每次咪挺的過程中總能獲益良多、獲得教授許多回饋豐富我的知識；也很謝謝教授的悉心指導，讓我碩士生活取得研究與生活平衡，不只有每天在做研究，還有許多的運動局，例如籃球、跑步、爬山，甚至還有實驗室旅遊及僅有一次的老師歸國二十週年 DVLab 旅行，十分感謝教授的用心良苦。

再來，我很感謝我的家人們總能做我最強的支柱和後盾，讓我能無後顧之憂的專心讀完碩士並完成此份論文，每當做到疲乏的時候，家裡永遠是最好的避風港、充電站，讓我能稍微回家喘口氣好好休息後繼續奮戰。

再來當然要感謝在這兩年中遇到的實驗室夥伴們，很謝謝 R10 的學長姐 Gordon、Jasmine、Arvind、Anita、嘉豪、謹譯、慕德，不論是修課上、研究上、或是人生規劃上，你們總是不厭其煩回答我的各種疑問。謝謝同屆 R11 的失業救濟戰友：文山兄弟黨 – 子瑜和彥儒老哥，以及竣哥，這兩年總是和你們打打鬧鬧吃吃喝喝，從碩零進來跟竣哥陌生，還要另外開群組討論怎麼回覆竣哥、到一起開始架構起 GV tool、中間一起跟教授去跑路跑、然後轉眼間到了碩二上一起焦慮



doi:10.6342/NTU202403664

找工作跟刷 leetcode、到碩二下換成焦慮論文，於是被推坑荒野亂鬥到現在三個月準備邁入無課一萬盃、還在最水深火熱的五月在兩次咪挺前去了一趟日本，很謝謝彥儒老哥在碩論最忙的五月排了一趟日本行，雖然文山兄弟黨都要退租還是沒去過你們家，但很謝謝你們讓我的碩班生活多了許多精彩。最後當然不能忘記感謝 R12 和 R13 的學弟妹們（雖然只有虹伶姐是學妹），謝謝 Wish、Bob、虹伶姐、建亨、瀞桓、亦翔、翔淳，常常也受到你們的幫助，不論是問問題、一起打 CAD 比賽、向你們要論文實驗需要的 design、講一些幹的、或是被你們詢問問題時都能收穫很多，尤其感謝翔淳和亦翔在最後一個實驗衝刺時幫了我很大的忙。雖然 R13 的學弟們只有在咪挺和實驗室宜蘭遊時遇到，但跟你們玩真的很好玩，尤其可以直接湊人十排荒野亂鬥真的很爽，之後一定有機會再見。

同時也很感謝助理 Sherry，謝謝 Sherry 和 Sherry 的先生偶爾投餵國外來的很酷的零嘴，讓每天都窩在實驗室的我能夠解嘴饞，也很謝謝 Sherry 對帳務的專業，讓實驗室每年的經費都能有條不紊的被使用，也讓我買到用起來很舒服的簡報筆、電競螢幕和機械鍵盤。

還有很謝謝跑步團 Ric、John、Wish、彥儒、瀞桓、昊晨，一起以路跑為目標，巔峰時期一週二到三練，讓我不只每天坐在電腦桌前而壞了身體健康，也讓我重新愛上了跑步，當成一個固定紓壓的管道，也在畢業前半年陸續達到了自己目標：台北馬初半馬破二 (109 分鐘)、台大校馬 5km 達到配速 4:17 (24 分鐘)、puma 10km 也跑進前百名 (48 分鐘)。

我也很謝謝這一路上願意錄取我當實習生的各公司主管們，讓我汲取業界經驗，也更好在做研究發想演算法時以實習經驗做過的事情套用進來。也很謝謝女朋友在我做研究壓力大到不行時會陪我出去透透氣放空沈澱一下，休息是為了走更長遠的路，沈靜過後總是能充飽電繼續重拾熱情衝刺研究。


doi:10.6342/NTU202403664

最後我一定要感謝的是竣哥的實驗室椅子，謝謝這個椅子讓我每天都能落實包包不落地政策，讓我的書包總能保持乾淨。

要感謝的人事物不勝枚舉，非常感謝你們在我的求學旅途中幫助、支持、鼓勵我，有你們才能讓我的碩班生活順利結束，願我們情誼長存。

陳孟宏 2024



doi:10.6342/NTU202403664

# 中文摘要

由於 IC 設計規模和複雜度的不斷增長，RTL 設計中的除錯變得越來越具有挑戰性。傳統上，工程師會生成測試模式並分析密集的波形文件，以追踪模擬結果，並與參考模型的正確結果進行比較。這個過程非常耗時且困難，因為涉及的信號眾多，包括輸入信號、輸出信號、內部信號和存儲器端口信號。

本論文介紹了我們的錯誤定位工具，可以自動過濾不相關的信號並突出顯示最有可能與錯誤相關的信號。通過根據信號的懷疑程度進行排序，並識別這些信號影響條件語句的控制路徑，該工具引導工程師找到最可能的問題區域。這顯著減少了除錯的時間和精力，使工程師能夠專注於最相關的信號和控制路徑，從而簡化了除錯密集波形和廣泛信號數據的複雜任務。這種方法將傳統上艱巨的 RTL 除錯任務轉變為更加可管理和高效的過程。

**關鍵字**：錯誤偵測、斷言、通用驗證方法學、驗證環境、積體電路設計



doi:10.6342/NTU202403664

# Abstract

Debugging in RTL design is increasingly challenging due to the growing scale and complexity of IC designs. Traditionally, designers generate test patterns and analyze dense waveform files to trace simulation results and compare them with the golden results from a reference model. This process is time-consuming and difficult because of the numerous signals involved, including input, output, internal, and memory port signals.

This thesis introduces a novel bug localization tool that automates the filtering of unrelated signals and highlights those most likely associated with bugs. By ranking signals based on their suspected degrees and identifying the control paths where these signals influence conditional statements, the tool guides designers to the most likely problem areas. This significantly reduces debugging time and effort by allowing designers to focus on the most relevant signals and control paths, thereby simplifying the complex task of debugging dense waveforms and extensive signal data. This approach transforms RTL debugging from a traditionally arduous task into a more manageable and efficient process.



doi:10.6342/NTU202403664

**Keywords:** Bug Diagnosing, Assertion, UVM, Verification Environment, Integrated Circuit Design



doi:10.6342/NTU202403664

# Contents





doi:10.6342/NTU202403664

doi:10.6342/NTU202403664

doi:10.6342/NTU202403664

# List of Figures





doi:10.6342/NTU202403664

doi:10.6342/NTU202403664

doi:10.6342/NTU202403664

# List of Tables





doi:10.6342/NTU202403664

# Chapter 1 Introduction

This chapter lays the groundwork for the thesis by presenting the problem statement and motivation, reviewing related works, highlighting the contributions of the research, and outlining the organization of the thesis. Section 1.1 defines the research problem and explains the motivation behind addressing this issue, emphasizing its significance in the field. In section 1.2, a review of existing literature and related works is provided, illustrating the current state of research and identifying gaps that this thesis aims to fill. In section 1.3, the key contributions of the research are outlined, showcasing the novel aspects and advancements introduced by this work. Section 1.4 provides a structured overview of the thesis and details the content and focus of each chapter to guide the reader through the document.

## 1.1 Problem Statement and Motivation

In the process of RTL design, designers inevitably encounter situations that require debugging. However, as the scale and complexity of IC designs increase, the debugging process becomes critically challenging. Typically, designers generate test patterns to target the RTL design, spend significant time tracing the simulation results in waveform files, and compare them with the golden results produced by a reference model written in a


doi:10.6342/NTU202403664

high-level language like C/C++ or Python. However, as shown in Fig. 1.1, waveforms are usually very dense (right figure) and contain numerous signals to be filtered (left figure), including input, output, internal, and memory port signals, making bug diagnosis and localization extremely difficult.

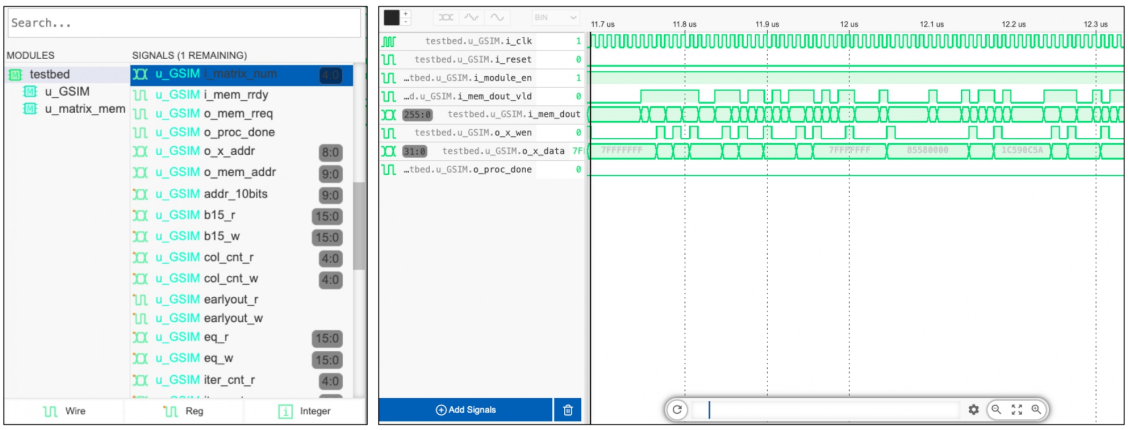



Figure 1.1: The example of debugging process **without** our tool

Hence, a bug localization tool that can automatically filter out unrelated signals and focus on a few highly probable signals would be extremely beneficial. This tool will rank signals by their suspected degrees and identify the control paths (e.g. always blocks, if/case statements) where these signal candidates control the conditional statements. Designers could then follow the tool's guidance to begin debugging from the most suspected zones, significantly reducing debugging time by narrowing down the search space for potential bugs. This thesis successfully develops such a bug localization tool. As illustrated in Fig. 1.2, unlike in Fig. 1.1, our tool allows designers to focus on the most suspected signals related to the bug's root cause (e.g. the signal $col\_cnt\_r$). Designers can start debugging by observing the behavior of these signals in the waveform file or examining the control path's coding logic within the RTL design. The tool also extracts and provides


doi:10.6342/NTU202403664

these control paths, significantly aiding designers in bug detection. This approach prevents designers from having to sift through dense waveforms and numerous signals from scratch, effectively eliminating the “needle in a haystack” problem.

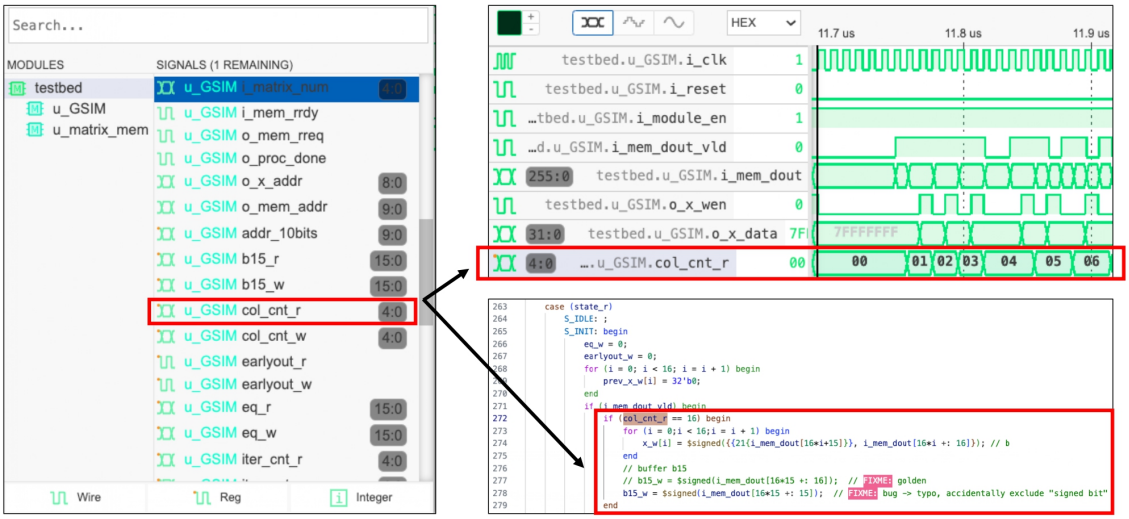
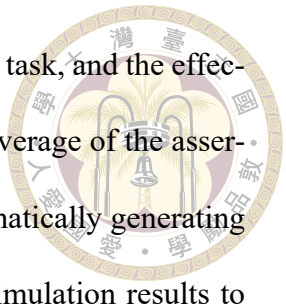



Figure 1.2: The example of debugging process **with** our tool

## 1.2 Related Works

In the IC design verification process, beyond the traditional methods of simulation-based and formal verification, there is an approach that enhances both: “assertion-based” verification [2]. By embedding assertions [3] within the design or testbench, these assertions are checked during simulation or formal verification to ensure the design behaves as expected, aiding in the early detection of errors by monitoring design behavior. Assertion-based verification leverages the strengths and mitigates the weaknesses of simulation-based and formal verification. For example, assertions can enhance simulation-based verification by catching errors early, while formal verification can be used to prove critical properties that are difficult to cover with simulation alone.


doi:10.6342/NTU202403664



However, writing assertions is a time-consuming and error-prone task, and the effectiveness of assertion-based verification depends on the quality and coverage of the assertions. To address this, many papers have proposed methods for automatically generating assertions. These include using decision tree-based algorithms on simulation results to mine assertions and applying formal verification to remove incorrect ones [4–7], using automatic test pattern generation (ATPG) algorithms [8], or abstracting between the RTL and its transaction-level model (TLM) for assertion-based verification [9, 10]. However, these methods often limit assertions to specific templates or applications, such as mining for security detection [11], and may not generate temporal assertions.

To overcome these limitations, other methods have been proposed, such as adopting any linear temporal logic (LTL) formula defined by designers, mining propositions from simulation traces, permuting combinations in designer-provided templates, and verifying correctness [12, 13]. Recently, large language models (LLMs) like GPT-4 have been used to generate and reason about assertions based on RTL design and designer specifications [14]. Although this approach is still immature and requires curated datasets, careful guidance, and significant costs for GPT-4 tokens, it represents a pioneering effort in assertion mining algorithms.

While these papers focus on automatically generating assertions, they do not consider the quality of the generated assertions or use this information to guide designers in bug localization. Therefore, another research focus is on proposing various metrics to evaluate and rank assertions based on different aspects, depending on the designer's intent. Assertion quality can be assessed by code coverage in RTL codes [15], activation frequency in simulation runs, correlation with other assertions through contingency tables [16, 17], belief-fail rate by analyzing absent scenarios in the simulation trace [18], or static analysis


doi:10.6342/NTU202403664

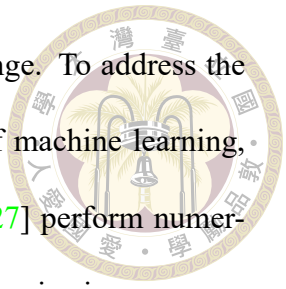
between the assertion and the RTL code to evaluate importance, complexity, and dominance [19, 20].

Although there are various approaches to verify RTL designs, including simulation-based, formal verification-based, and assertion-based methods, effectively utilizing verification results to automatically detect and localize potential bugs is a critical challenge in modern IC development due to increasing design complexity and scalability. Simulation-based verification, while widely used, is inherently incomplete because it is impractical to exhaustively simulate all possible patterns. Formal techniques like model checking promise to address this issue through comprehensive state-space traversal but suffer from scalability issues with complex designs.

Several works have focused on automatic bug localization, which can be categorized into three main approaches: analysis-based, ML-based, and assertion-based. In analytical methods, some approaches [21] prioritize suspected bug locations by analyzing whether control path statements affect erroneous output signals or whether masking the data path impacts the erroneous output, calculating a confidence score for RTL lines. However, this method is only suitable for small, specific designs. Other papers [22, 23] use dynamic approaches, applying simulation patterns and extracting erroneous results compared to a golden reference, then forming a constraint formula for the unexpected result and using word-level satisfiability (SAT) solvers to identify potential bug candidates, but they encounter similar limitations as [21]. Additionally, [24] combines model checking with Bayesian networks to analyze counterexample hints and guide potential buggy subregions.

Machine learning techniques have also been proposed to predict possible bug locations [25, 26], but collecting sufficient training data of RTL designs with inserted bugs


doi:10.6342/NTU202403664



that cover numerous designs and bug types remains a critical challenge. To address the scalability issues of formal techniques and the accuracy concerns of machine learning, assertion-based bug localization offers a compromise. Papers like [27] perform numerous simulation runs to generate assertions based on simulation traces, viewing common assertions as the most suspected candidates and mapping them back to RTL codes. [28] mines assertions from passing simulation traces, treating them as golden assertions, and uses these with additional simulations to localize possible bug zones by analyzing where assertions fail. [1] enhances counterexamples by inserting redundant propositions into assertions from failed simulation traces, ranking these suspected assertions and mapping them back to RTL codes by analyzing the control paths with antecedent signals.

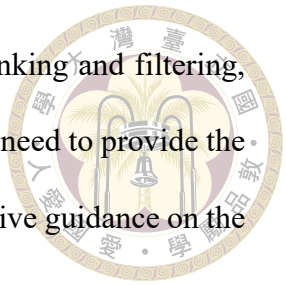
However, these papers often rely solely on information from the buggy design and use intuitive methods to determine whether the mined assertions are golden or suspected, which may be inaccurate without considering the actual specification or reference model behavior. Therefore, in the next section, we will explore the new directions our work addresses, highlighting the advantages and contributions it brings to the field.

## 1.3 Contributions

In our work, we aim to provide an end-to-end automated approach for bug localization by generating assertions for the target design and identifying the most suspected bug positions, offering guidance to designers to assist in the debugging process. Our contributions are as follows:

1. **Development of a comprehensive verification flow**: We have created an end-to-end flow for building a verification environment aimed at bug localization. This


doi:10.6342/NTU202403664



flow encompasses automatic assertion generation, assertion ranking and filtering, and bug localization within RTL code sections. Designers only need to provide the target design and basic settings, as detailed in Chapter 4, to receive guidance on the most suspected bug positions to begin their debugging process.

2. **Adaptability to various RTL designs**: Our approach is adaptable to a wide range of RTL designs, not just specific ones like the RISC-V processor as seen in previous works. This provides flexibility and broad applicability in the verification process.

3. **Handling diverse bug types**: Beyond generalizing RTL designs, our method is also versatile with respect to different types of bugs. It is not restricted to simple mutations (e.g. changing "<" to ">") and is applicable to both safety and liveness bugs, as discussed in Chapter 2.

4. **Accurate bug localization hints**: We have developed a metric to evaluate and rank suspected signals that may be causing the bug. Experiments demonstrate our ability to accurately identify the signal with the highest suspected score, as elaborated in Chapter 8.3.

Since our work is an end-to-end tool covering four phases—preparation, assertion generation, assertion validation, and bug localization—each phase can be explored as a separate research direction. To streamline our entire workflow, we utilize several off-the-shelf third-party tools. Below, we highlight the original contributions of our work:

1. **Configuration settings in the preparation phase**: We design a series of parameters to build a verification environment for general RTL designs, not limited to specific types such as ALU, MIPS, or RISC-V.


doi:10.6342/NTU202403664

2. **Constrained random simulation method in the assertion validation phase**: After automatically generating numerous assertions using a third-party tool [29], we propose a constrained random simulation method to select only hundreds of useful assertions from tens of thousands, aiding our bug localization process.

3. **Metrics in the bug localization phase**: After selecting the useful assertions, we map them back to the RTL code [1]. We design a metric to calculate the suspicious score related to bugs for each controlling signal, and locate the bug position on the control path of the most suspected controlling signal.

## 1.4 Thesis Organization

This thesis is organized into four main parts: introduction and background (Chapters 1-2), framework overview (Chapter 3), detailed component analysis (Chapters 4-7), and results with future directions (Chapters 8-9).

Chapter 1 provides an overview of the research problem, objectives, and the significance of the study.

In Chapter 2, essential background information and related work are discussed to provide context and foundation for the research.

While in Chapter 3, a comprehensive explanation of our bug localization framework is presented, outlining the overall methodology and approach.

Chapter 4 describes the initial setup process, including how to configure the files necessary for the framework.

In Chapter 5, detailed methods for generating valid assertions from the Design Under


doi:10.6342/NTU202403664

Verification (DUV) are explained.

In Chapter 6, techniques for filtering and validating suspected assertions to narrow down potential bugs are discussed.

In Chapter 7, the process of localizing bugs within RTL codes using the filtered assertions is elaborated.

Chapter 8 presents the experimental findings, demonstrating the effectiveness of the proposed methods.

And finally Chapter 9 summarizes the research contributions and discusses potential future directions for further improvement and research.



doi:10.6342/NTU202403664

# Chapter 2 Background Knowledge

In this chapter, we provide the necessary background knowledge for understanding the subsequent work. First, we explain several key terms that will be frequently used throughout the thesis (Section 2.1). Then, we introduce assertions and assertion-based verification in Sections 2.2 and 2.3, which are the core concepts of the thesis. Next, we elaborate on the architecture of the Universal Verification Methodology (UVM), a standardized methodology for verifying IC designs, where our work utilizes similar concepts to build the verification process (Section 2.4). Finally, we discuss the third-party tools used in our flow (Section 2.5).

## 2.1 Terminology Definitions

This section provides comprehensive definitions for key terms used throughout the thesis.

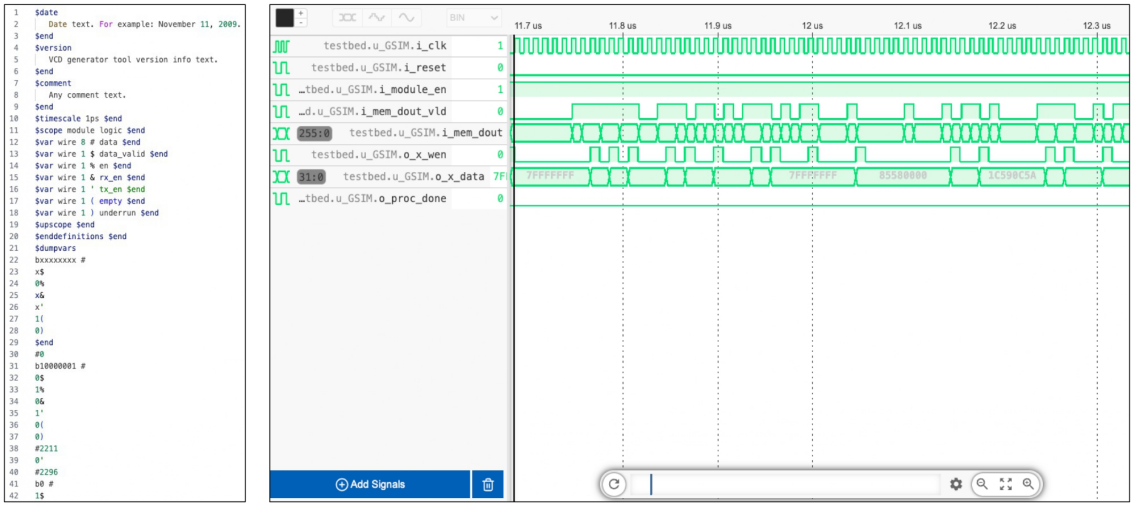
**Definition 2.1.1 (Design Under Verification (DUV)).** The DUV is the specific IC design or module being tested and verified to ensure it functions correctly according to its specifications. Specifically, it refers to the target design in the verification process.

**Definition 2.1.2 (Reference Model).** A reference model is a high-level, often abstract, version of the design that serves as a correct behavioral standard against which the DUV


doi:10.6342/NTU202403664

is compared during verification.

**Definition 2.1.3 (Value Change Dump (VCD)).** A VCD file is a standard format for recording the activity of digital signals in a simulation. It logs changes in signal values over time, helping designers analyze the behavior of the DUV. The example of a VCD file in text format and its visualization as a waveform is illustrated in Fig. 2.1.




(a) (b)

Figure 2.1: The example of VCD file. (a) text format, (b) visualization as a waveform

**Definition 2.1.4 (Liveness Bug).** A liveness bug in RTL design indicates something good fails to happen (system does not make progress or complete tasks), which can be localized by our tool.

**Definition 2.1.5 (Safety Bug).** A safety bug in RTL design indicates something bad happens (system enters an incorrect or hazardous state), which can also be localized by our tool.

**Definition 2.1.6 (Control Path Block).** These are structures within an RTL design that dictate the flow of control in the circuit. Examples include:


doi:10.6342/NTU202403664

- **Always Block**: Executes its statements whenever a specified condition or event occurs.

- **If Statement**: Executes a block of code based on whether a condition is true or false.

- **Case Statement**: Selects and executes one block of code from multiple options based on the value of an expression.

## 2.2 SystemVerilog Assertions

SystemVerilog Assertions (SVAs) are a powerful feature of the SystemVerilog hardware description and verification language [30]. SVAs enable designers to specify expected behaviors and properties of their digital designs formally. By embedding these assertions directly into the design or testbench, designers can automatically check that the design adheres to these specified behaviors during simulation or formal verification processes.

Assertions can be classified into two main types, as illustrated in Fig. 2.2:

- **Immediate Assertions**: These are checked at the moment they are encountered during simulation. They are typically used for simple checks within procedural code. We can view it as a *if* statement.

- **Concurrent Assertions**: These are continuously monitored throughout simulation time. They are used to specify complex temporal properties and sequences of events.


doi:10.6342/NTU202403664

```
1  assert (A == B) $display ("OK. A equals B");
2    else $error("It's gone wrong");
```

(a)

```
1  assert property (!(Read && Write));
```

```
1  assert property (@(posedge Clock) Req |-> ##[1:2] Ack);
```

(b)

Figure 2.2: The example of the (a) immediate assertion, and (b) concurrent assertions

Using SystemVerilog Assertions helps in early detection and diagnosis of design errors, improving the reliability and robustness of the verification process.

## 2.3 Verification Approaches for RTL Designs

In this section, we delve into the various verification approaches used in RTL design. Verification is a critical step in the design process, ensuring that the design functions correctly and meets its specifications. We will explore three primary methods: simulation-based verification, formal-based verification, and assertion-based verification. Each of these methods has its unique strengths and limitations, and understanding them is essential for effective and efficient verification of complex IC designs.


doi:10.6342/NTU202403664

### 2.3.1 Simulation-Based Verification

Simulation-based verification is the most widely used method for verifying RTL designs. It involves simulating the design with a series of test vectors to check for correct functionality. This approach mimics the behavior of the design under different scenarios by applying input stimuli and observing the outputs.

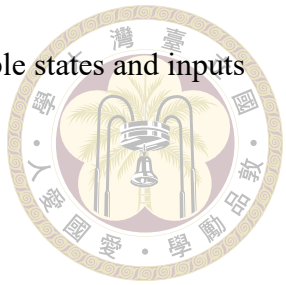
- **Key Features**:
  - **Dynamic Nature**: Simulation checks the design's behavior over time, allowing designers to observe how the design responds to a sequence of inputs.
  - **Flexibility**: It can handle complex scenarios and interactions within the design, providing detailed insights into the design's performance.
  - **Visualization**: Designers can visualize the simulation results using waveform viewers, which help in diagnosing and debugging issues.
- **Limitations**:
  - **Coverage**: Achieving exhaustive coverage can be challenging, as it is impractical to simulate all possible input combinations and states.
  - **Time-Consuming**: Simulations can be slow, especially for large and complex designs, making it difficult to verify all aspects of the design comprehensively.

### 2.3.2 Formal-Based Verification

Formal-based verification uses mathematical methods to prove the correctness of a design with respect to its specifications. Unlike simulation, which tests the design against


doi:10.6342/NTU202403664

a set of test cases, formal verification exhaustively explores all possible states and inputs to ensure the design' s correctness.



- **Key Features**:
  - **Exhaustiveness**: Formal verification can provide complete coverage by exploring all possible states and input combinations.
  - **Proof of Correctness**: It can formally prove that certain properties hold true for the design, providing a higher level of assurance.
  - **Scalability Issues**: Although powerful, formal methods can struggle with scalability, especially for large designs with complex state spaces.
- **Limitations**:
  - **Complexity**: Setting up formal verification requires a deep understanding of both the design and formal methods, making it more complex than simulation-based verification.
  - **Resource-Intensive**: Formal verification can be computationally expensive, requiring significant time and computational resources.

### 2.3.3 Assertion-Based Verification

Assertion-based verification combines elements of both simulation and formal verification. It involves embedding assertions, which are statements of expected behavior, directly into the design or testbench. These assertions are then checked dynamically during simulation or statically during formal verification.

- **Key Features**:


doi:10.6342/NTU202403664

  - **Early Detection**: Assertions help in early detection of design errors by continuously monitoring the design's behavior against specified properties.
  - **Reusability**: Assertions can be reused across different verification environments, improving verification efficiency.
  - **Complementary**: It complements both simulation and formal methods by providing a mechanism to check specific properties during the verification process.

- **Limitations**:
  - **Writing Assertions**: The effectiveness of assertion-based verification depends on the quality and coverage of the assertions, which can be time-consuming and error-prone to write.
  - **Tool Support**: Effective assertion-based verification requires good tool support for writing, checking, and debugging assertions.

## 2.4 UVM

Universal Verification Methodology (UVM) is an advanced verification methodology for verifying IC designs [31, 32]. It is a standardized, comprehensive framework developed by Accellera, an industry consortium, to address the challenges associated with verifying complex designs in a systematic and scalable manner. UVM leverages object-oriented programming principles and builds on top of the SystemVerilog language to provide a robust and reusable verification environment. The basic architecture of UVM is shown in Fig. 2.3


doi:10.6342/NTU202403664

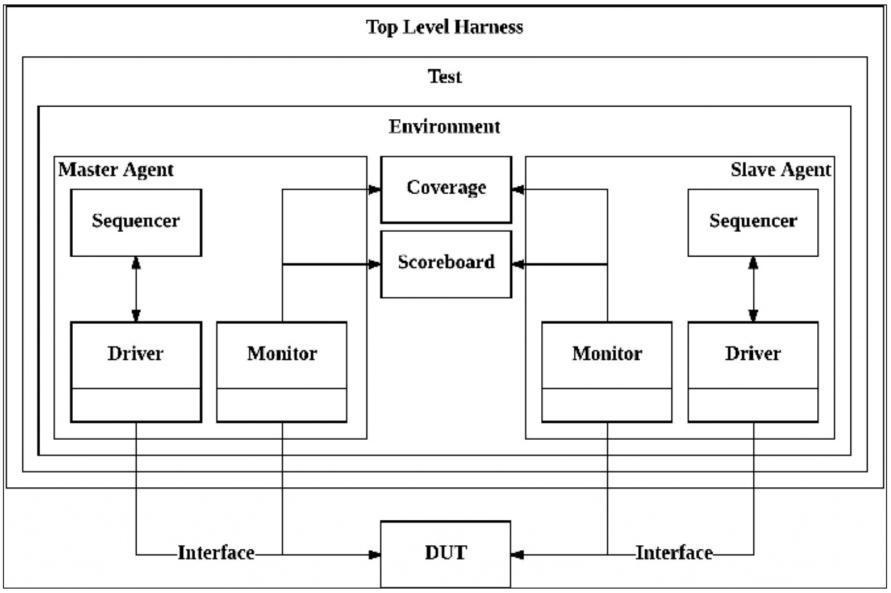



Figure 2.3: UVM’ s fundamental architecture

- **Key Components of UVM**

  1. **Verification Components**:

     – **Driver**: Drives stimulus to the Design Under Verification (DUV).

     – **Monitor**: Observes the DUV’s outputs and collects data for analysis.

     – **Sequencer**: Manages the sequence of operations and drives the data through the driver.

     – **Agent**: Combines driver, monitor, and sequencer to encapsulate a complete verification environment for a specific interface or protocol.

  2. **Testbench Structure**:

     – **Environment**: A container for all the components of the testbench, including agents, scoreboards, and coverage collectors.

     – **Test**: Specifies the stimulus and configuration for the environment.

     – **Sequence**: A series of transactions or operations applied to the DUV.

     – **Scoreboard**: Compares the actual output of the DUV with the expected


doi:10.6342/NTU202403664

output to check for correctness.

3. **Configuration and Factory**:
   - **Configuration**: UVM allows for easy configuration and reconfiguration of the testbench components without modifying the code.
   - **Factory**: A mechanism for creating objects dynamically, enabling greater flexibility and reuse of components.
4. **Phases**: UVM divides the verification process into well-defined phases, such as build, connect, run, and cleanup phases, to ensure orderly and predictable execution of the testbench.
5. **Transaction-Level Modeling (TLM)**: TLM allows communication between components at a high level of abstraction, improving simulation speed and simplifying the testbench architecture.

- **Benefits of UVM**

1. **Reusability**: UVM promotes reusability of verification components across different projects and designs, reducing development time and effort.
2. **Scalability**: UVM's modular approach and standardized structure make it scalable for complex designs, allowing for easy expansion and maintenance of the verification environment.
3. **Consistency**: By following a standardized methodology, UVM ensures consistency in verification practices across different teams and projects, facilitating better collaboration and understanding.
4. **Efficiency**: UVM's automation features, such as the factory and configuration mechanisms, enhance verification efficiency by reducing manual intervention


doi:10.6342/NTU202403664

and enabling rapid iteration and testing.

5. **Comprehensive Coverage**: UVM supports advanced verification techniques like constrained random stimulus generation, functional coverage, and assertions, ensuring thorough and robust verification of the DUV.

- **Example Workflow in UVM**

1. **Testbench Development**:
   - Define the testbench architecture using UVM components such as drivers, monitors, sequencers, and agents.
   - Implement the DUV interface and connect the UVM components to the DUV.
2. **Configuration**:
   - Configure the testbench components and specify the desired verification scenarios.
   - Use the UVM configuration database to manage configuration settings dynamically.
3. **Stimulus Generation**:
   - Develop sequences to generate the desired stimulus for the DUV.
   - Use constrained random generation and directed tests to cover different aspects of the design.
4. **Simulation and Analysis**:
   - Run simulations and collect coverage data.
   - Use the scoreboard and functional coverage collectors to analyze the DUV's behavior and ensure it meets the specifications.


doi:10.6342/NTU202403664

5. **Debugging and Refinement**:

   – Identify and debug issues using UVM's rich set of debugging tools and features.

   – Refine the testbench and stimulus based on the results to achieve higher coverage and validation confidence.

By providing a detailed and systematic approach to verification, UVM plays a crucial role in ensuring the quality and reliability of modern IC designs. Hence, in our tool, we incorporate UVM concepts into our algorithm. For example, we generate constrained random patterns and apply these to both the DUV and the reference model, similar to the functions of the "sequencer" and "driver" in UVM. We then extract the valid output results from the DUV and compare them with the reference model's results for further analysis, akin to the roles of the "monitor" and "scoreboard" in UVM. By utilizing the UVM architecture, our tool constructs a complete verification environment that effectively supports the bug localization process.

## 2.5 Third Party Tools Utilized in Our Flow

In this section, we introduce the third-party tools integrated into our verification framework to enhance the efficiency and accuracy of the bug localization process. Each of these tools plays a vital role, contributing unique capabilities that complement our methodology. By leveraging these specialized tools, we create a robust and comprehensive verification environment. The following sections provide detailed descriptions of each tool and its specific function within our framework.


doi:10.6342/NTU202403664

### 2.5.1 SDM—Output Matcher

SDM (Sequential Design Matching) [33] is a tool designed to analyze waveform files by comparing temporal patterns of specific signal occurrences. It superimposes correct and erroneous waveforms, identifies potential error locations, and generates a new consolidated waveform file. This allows engineers to use standard waveform viewers for quick error identification and correction. SDM is integral in validating the DUV by comparing its behavior with a reference model, enabling efficient error detection and localization.

### 2.5.2 HARM—Assertion Miner

HARM (Hint-Based Assertion Miner) [29, 34] is a tool crafted to produce linear temporal logic (LTL) assertions using user-specified hints and simulation traces from the DUV. It operates without requiring access to the DUV source code, relying exclusively on simulation data. By leveraging user-provided LTL templates, propositions, and ranking metrics, HARM refines its search space to generate high-quality assertions. This capability empowers verification engineers to integrate their expertise into the automated assertion generation process, enhancing verification efficiency and accuracy significantly.

### 2.5.3 Pyverilog—Code Analyzer

Pyverilog [35, 36] is an open-source hardware design and analysis framework written in Python. It provides a powerful toolset for parsing, analyzing, and transforming Verilog HDL (Hardware Description Language) code. In our verification framework, Pyverilog is utilized as a code analyzer to parse the RTL design, extract essential information, and perform static analysis. This tool helps in understanding the design structure, identify-


doi:10.6342/NTU202403664

ing control paths, and generating intermediate representations that are crucial for further processing in the bug localization workflow.

### 2.5.4 Icarus Verilog—Simulator

Icarus Verilog [37, 38] is an open-source Verilog simulation and synthesis tool. It supports a wide range of Verilog standards and is widely used for simulation purposes. In our framework, Icarus Verilog serves as one of the primary simulators for running test patterns against the DUV. It generates simulation results that are used to compare the behavior of the DUV with the reference model, facilitating the identification of discrepancies and potential bugs. Its open-source nature and flexibility make it a valuable tool in our verification process.

### 2.5.5 Synopsys VCS—Simulator

Synopsys VCS (Verilog Compiler Simulator) [39, 40] is a highly advanced and widely used commercial simulator for RTL designs. It offers high-performance simulation capabilities, comprehensive debug features, and support for a variety of verification methodologies. In our verification framework, VCS is employed to perform detailed and high-fidelity simulations of the DUV. It helps in generating accurate simulation results that are essential for verifying the design's correctness and for subsequent bug localization steps. VCS's robust features and industry-standard reliability make it a critical component of our verification toolkit.


doi:10.6342/NTU202403664

### 2.5.6 Synopsys VC Formal—Formal Verifier

Synopsys VC Formal [41] is a formal verification tool that uses mathematical techniques to prove the correctness of digital designs. Unlike simulation-based approaches, formal verification exhaustively explores all possible states and scenarios to ensure that the design adheres to its specifications. In our framework, VC Formal is utilized to formally verify the assertions generated during the bug localization process. It helps in proving or disproving these assertions, providing a higher level of confidence in the design's correctness. VC Formal's rigorous analysis capabilities make it indispensable for ensuring thorough verification and robust bug localization.


doi:10.6342/NTU202403664

# Chapter 3 Overview of the Bug Localization Flow

In this chapter, we begin by exploring how our tool intuitively guides designers through the debugging process (Section 3.1). Following this, we delve into the architecture of our framework, detailing the interplay between its various components (Section 3.2). Also, we examine the intricate relationship between the design and generated assertions, highlighting the critical role assertions play in aiding the bug localization process (Section 3.3). Finally, we make assumptions (Section 3.4) and categorize (Section 3.5) target designs for our tool.

## 3.1 How Our Tool Guides Designers to Debug

Our tool guides designers through a structured debugging process designed to enhance efficiency and accuracy. The flow in Fig. 3.1 begins with designers' setting up the configuration file and tailoring the tool to their specific design requirements. Following this initial setup, our tool takes over. First, it automatically generates a large number of assertions to address issues that would be time-consuming and error-prone to write manually. Then, it filters these assertions to isolate valid ones, further pinpointing potential


doi:10.6342/NTU202403664

issues within the design.

Once the assertions have been processed, our tool identifies the most likely location of the bug, providing designers with a targeted area to investigate. This pinpointed approach significantly reduces the time and effort needed to locate and understand the root cause of the problem. Finally, designers use this information to fix the bug in the RTL code, completing the cycle of detection and correction. This systematic flow not only accelerates the debugging process but also enhances the reliability and performance of RTL designs.

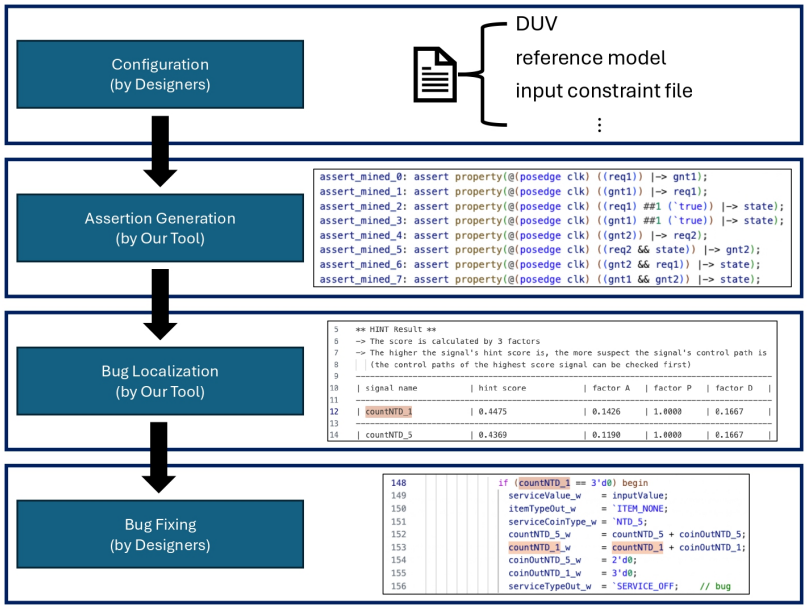



Figure 3.1: The flow of guiding designers to do bug localization with our tool

## 3.2 The Architecture of Our Framework

In this section, we will provide an overview of our framework, which is composed of four major parts as shown in Fig. 3.2.

1. **Preparation Phase—Configuration File**: The primary goal of this phase is to configure the settings for the subsequent phases. To start, designers should set the


doi:10.6342/NTU202403664

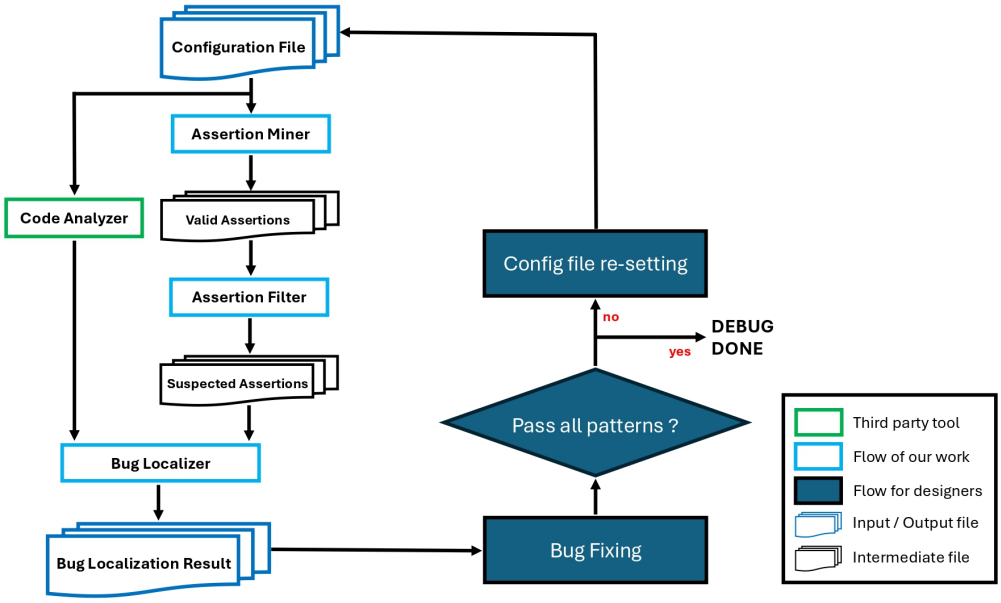



Figure 3.2: The overview of our framework

configuration file, as shown at the top of Fig. 3.2. This file contains crucial information for our tool to build the verification environment, including the path to the DUV, the module name to be verified, the clock signal name, whether the reset signal is negative or positive edge-triggered, and other more complex settings. The configuration file template is shown in Fig. 3.3, with detailed explanations provided in Chapter 4.

2. **Assertion Generation Phase - Assertion Miner**: With the configured settings, our objective in this stage is to automatically generate numerous valid assertions to comprehensively outline the operational scope of the DUV. The definition of *valid assertions* will be clarified later. As depicted in Fig. 3.4, we first use the third-party tool SDM to align output signals between the DUV and the reference model. Subsequently, our tool extracts and preprocesses the necessary information. We then utilize another third-party tool, HARM, which automatically extracts assertions from the DUV based on simulation results provided in the configuration file. These ex-


doi:10.6342/NTU202403664


Figure 3.3: The configuration file template

tracted assertions are integrated into the DUV and subsequently analyzed by another third-party tool VC Formal. This tool categorizes the assertions into *proven*, *falsified*, and *vacuous* states using formal verification techniques. From these, we select assertions in the proven state as our candidates for valid assertions, focusing only on correct signal implications relevant to the DUV for our bug localization process. Further details will be elaborated in Chapter 5.

3. **Assertion Validation Phase—Assertion Filter**: In the Assertion Generation Phase, we produce many valid assertions as candidates, but they are not necessarily correct according to the reference model or the spec. Because the DUV may contain bugs relative to the reference model, assertions generated from the DUV's simulation results may also be flawed with respect to the reference model, even if they appear correct for the DUV. Therefore, the primary goal at this stage is to further filter and validate these assertions to determine which ones are genuinely valid for both the DUV and the reference model, and which are suspected, being valid for the


doi:10.6342/NTU202403664

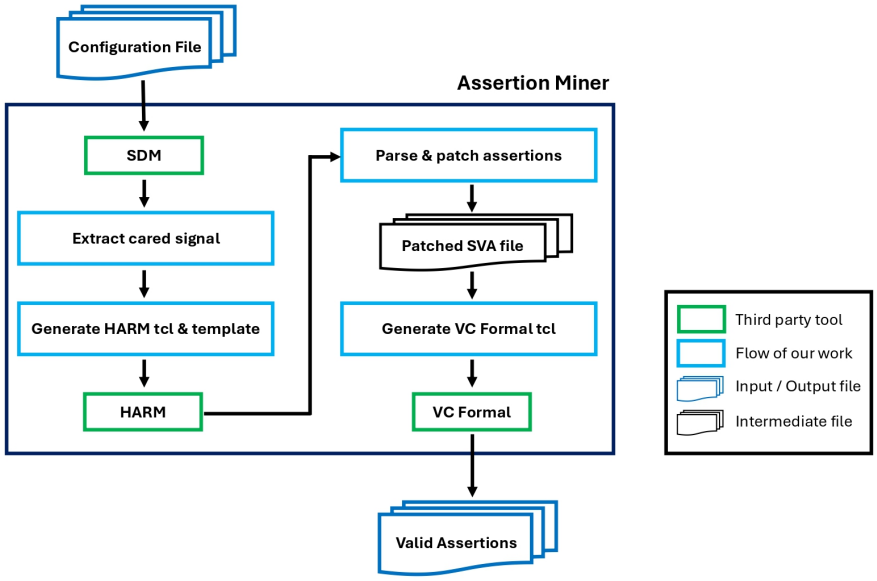



Figure 3.4: The detailed flow of Assertion Miner

DUV but failing against the reference model. Our algorithm retains these suspected assertions for the final phase of bug localization. The rationale behind this approach will be discussed in Section 3.3, and the detailed algorithms for this stage will be explored in Chapter 6.

4. **Bug Localization Phase—Bug Localizer**: Before proceeding, let's discuss the final results we will provide to designers. As shown in Fig. 3.5, we offer two types of guidance to assist in debugging. First, the *Sorted Hint Signals* (left figure) lists a few signals, sorted by the likelihood of a bug appearing in the control path conditional statements associated with these signals. Second, the *Suspect Control Paths* (right figure) highlights specific code sections under suspect control paths for designers to check in sequence.

   Referring back to Fig. 3.2, after obtaining the final assertion candidates from the Assertion Validation Phase, we use the filtering information of signals in the assertions' antecedents to calculate a hint score for each signal. The higher a signal's


doi:10.6342/NTU202403664

hint score, the more likely a bug is present in its control path, forming the basis of the Sorted Hint Signals guidance.

Additionally, with the aid of Pyverilog, we extract the abstract syntax tree (AST) of the DUV. This allows us to identify line numbers, signals, assigned values, and other control path information. By combining this structural information with the signal hint scores, we can pinpoint Suspect Control Paths for designers, directing them to begin their debugging efforts there. Detailed explanations will be provided in Chapter 7.

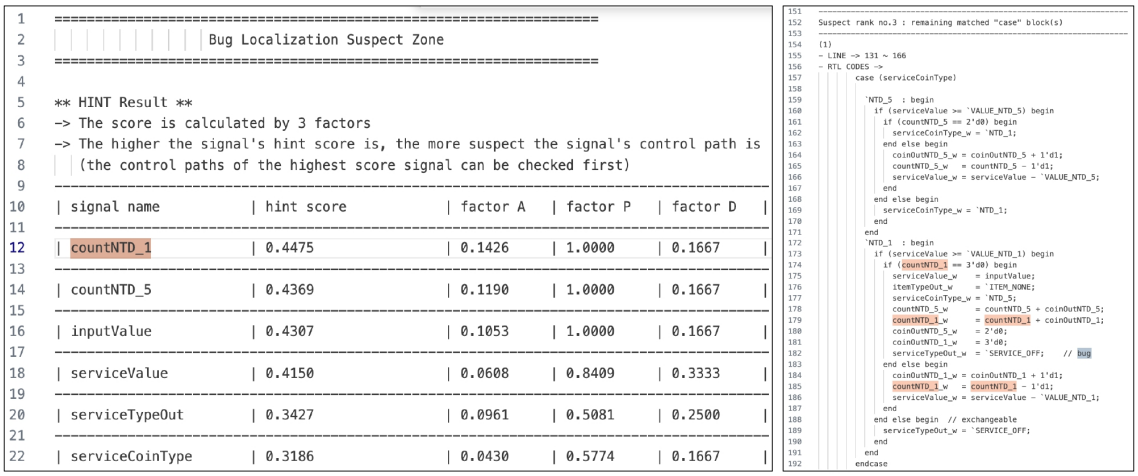



Figure 3.5: Demonstration of bug localization results: Sorted Hint Signals and Suspect Control Paths

## 3.3 Relation between the Design and Assertions

In Section 3.2 (Assertion Validation Phase), we noted that assertions in the proven state from VC Formal would serve as our valid assertion candidates. In this section, we will explain the relationship between the DUV, the reference model, and these valid as-


doi:10.6342/NTU202403664

sertion candidates to clarify the rationale behind this choice.

As depicted in Fig. 3.6, we assume the state space of the DUV is represented by the black circle (though we cannot precisely outline this space). The state space of the reference model is represented by red circles, which may overlap with the DUV's space. The overlapping area indicates shared functionality: starting from the same initial state and given the same input set, both the DUV and the reference model will transit to the same next state, producing identical output results (from point *A* to point *B* in Fig. 3.6).

For the assertions mined from HARM, we verify their correctness regarding the DUV using VC Formal, which categorizes them into *proven*, *falsified*, and *vacuous* states. Here, we focus on the *proven* and *falsified* states. Assertions in the falsified state (green circle) do not encompass the entire DUV state space, as a falsified status in formal verification means the DUV eventually reaches a counterexample for the assertion. Consequently, we cannot guarantee the relationship between the DUV and the falsified assertions, so we do not consider them as valid assertion candidates. However, assertions in the proven state (blue circle) consistently cover the DUV state space. Although we cannot determine the relationship between the reference model's state space and the proven assertions' state space (as shown in the left and right parts of Fig. 3.6), we can still utilize the proven assertions for the bug localization process.

If there is a bug in the DUV, starting from the same state and given an input set, the DUV will transit to a different state compared to the reference model. For instance, starting from the point *A* with a given input set, the reference model should transit to the point *C*, but the DUV transits to the point *D*. This unexpected behavior might be caused by a bug. Therefore, we will further examine this unexpected behavior using the waveforms


doi:10.6342/NTU202403664

of assertions in the proven state that intersect with the unexpected waveform. Detailed explanations will be provided in Chapter 6.

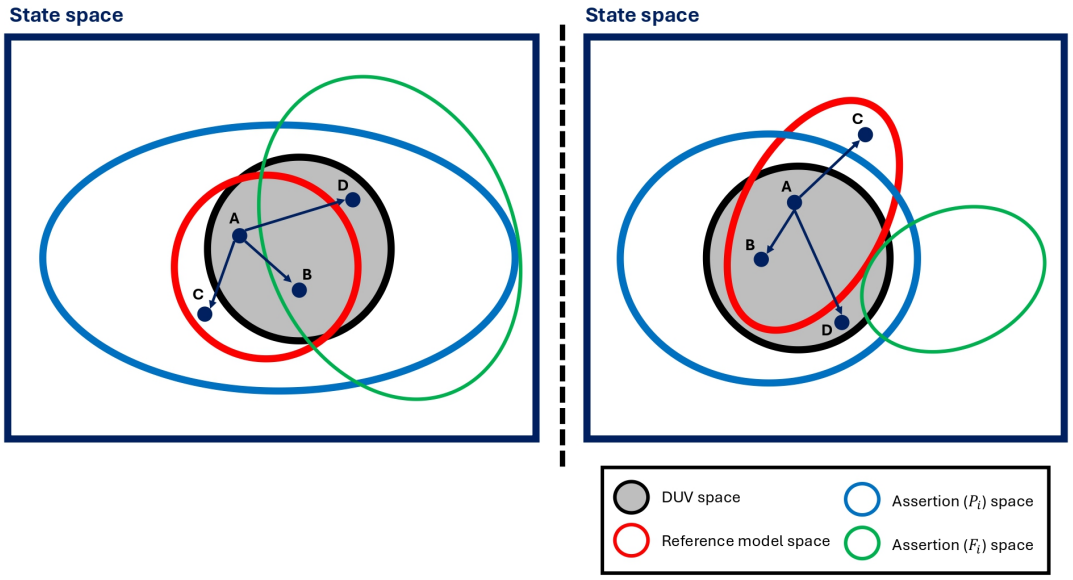



Figure 3.6: The relationship between the DUV, the reference model, and assertion candidates

## 3.4 Assumptions to the Design

In this section, we will outline the key assumptions underlying our algorithm with respect to the target DUV and its corresponding reference model. These assumptions are critical for ensuring the accuracy and effectiveness of our bug localization tool. Understanding these foundational premises will provide clarity on the conditions and constraints within which our tool operates.

For the DUV, the following assumptions are made based on its description language, circuit type, functionality, and input-output ports.

1. **The DUV should be implemented in Verilog or SystemVerilog**: In our implementation, most random simulation processes use the open-source third party tool


doi:10.6342/NTU202403664

Icarus Verilog as simulator. This tool primarily supports (System)Verilog applications and has limited support for VHDL and other hardware description languages. Therefore, our tool handles only DUVs written in (System)Verilog.

2. **The DUV should be a sequential design**: In our work, we will not address the bug localization flow in combinational designs, as another algorithm, called Engineering Change Order (ECO), is used to identify and correct rectification points in combinational DUVs. This topic is beyond the scope of our discussion.

3. **The DUV and the reference model should perform the same task**: The reference model serves as a golden reference during the verification process. Ensuring it matches the RTL design allows for accurate comparison and validation, helping to identify discrepancies and bugs early in the design cycle. Besides, consistency between the two ensures that simulations in our flow provide meaningful insights and effective bug localization.

4. **The DUV should have the same input ports as the reference model**: In our simulation process, we assume that the DUV operates on the same input pattern as the reference model. This means both should have identical input ports, including the width and order of each input signal.

While for the reference model, We **do not** require it to have a similar structure to the DUV; instead, we treat it as a black box that operates on the same set of inputs as the DUV and outputs the expected results. The reference model can be implemented in any language, from hardware description languages like SystemVerilog to high-level languages such as C/C++ or Python. Additionally, we **do not** expect the reference model to follow any naming conventions related to the DUV, nor do we require it to match the


doi:10.6342/NTU202403664

DUV's time series cycle-by-cycle.

1. **The functionalities of the reference model should satisfy the spec**: The reference model encapsulates the intended behavior and functionalities of the design. Ensuring the RTL design matches the reference model guarantees that the final hardware adheres to the specified requirements and behaves as intended.

2. **The reference model is considered bug-free**: The reference model serves as the gold standard against which the RTL design is verified. If the reference model contains bugs, it cannot provide a reliable baseline, leading to incorrect conclusions about the RTL design's correctness. Hence, considering the reference model as bug-free is essential for reliable verification, efficient debugging, and trustworthy simulation results.

## 3.5 Basic Categories for RTL Designs

To meet the needs of our tool, we categorize the RTL designs into several aspects, each corresponding to specific parts of the configuration settings which will be discussed in Chapter 4. The following sections will introduce the specific cases we need to address:

1. **Input**:

   - **Whether there exists (an) input valid signal to indicate the readiness of input data**: In certain designs, there may be no input valid signal indicating the readiness of input data, whereas other designs incorporate at least one input valid signal that can be set high to indicate the validity of the input data. The sample waveforms in Fig. 3.7 illustrate different scenarios: from top to bottom,


doi:10.6342/NTU202403664

they depict cases *without an input valid signal*, *with one input valid signal*, and *with more than one input valid signal*.


Figure 3.7: The sample waveforms illustrating the input valid signal

- **Whether the data requires more than one cycle to input**: In certain designs, input data is processed in a single cycle, indicating that the input valid signal, if present, remains active for only one cycle, as depicted in the top one of Fig. 3.7. Conversely, other designs input data over multiple cycles, as shown in the center and the bottom example of Fig. 3.7.

- **The spec defines constraints between input signals**: In some designs, certain input signals must adhere to constraints outlined in the spec. For instance, input points might need to be non-collinear. These constraints will be specified in the configuration file discussed in Chapter 4, and we will utilize the third-party tool VCS to handle these requirements.

- **The format of input patterns provided by the designer to the DUV differs from the format defined by our tool**: In our tool, we establish a standard-


doi:10.6342/NTU202403664

ized input format where the order of input signals in the pattern follows the declaration order in the DUV. Detailed guidelines will be provided in Chapter 4. However, variations in the designer's input format, such as different order of signals in input patterns, may lead to errors during random pattern generation and automatic input to the DUV. Therefore, in Chapter 4, we will discuss an approach akin to the adapter used in UVM architecture to address this challenge.

2. **Output**:

   - **Whether there exists (an) output valid signal to indicate the readiness of output results**: In certain designs, there may be no output valid signal indicating the readiness of output results, whereas other designs incorporate at least one output valid signal that can be set high to indicate the validity of the output results. The sample waveforms in Fig. 3.8 illustrate different scenarios: from top to bottom, they depict cases *without an output valid signal*, *with one output valid signal*, and *with more than one output valid signal*.
   - **Whether the results requires more than one cycle to output**: In some designs, the output results are generated in a single cycle, meaning the output valid signal, if present, lasts for only one cycle, as depicted in the bottom one of Fig. 3.8. Conversely, other designs output results over multiple cycles, as shown in the center example of Fig. 3.8.

3. **Clock**: In our work, we focus solely on synchronous designs due to the limitations of the open-source third-party tools we use. Consequently, we do not handle designs that involve different clock domains, clock gating, or other asynchronous techniques.


doi:10.6342/NTU202403664

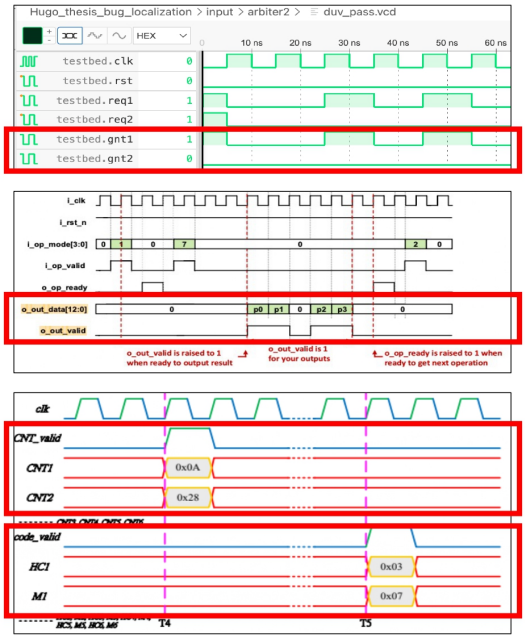



Figure 3.8: The sample waveforms illustrating the output valid signal

4. **Memory**:

- **The DUV contains SRAM**: SRAM in an RTL design is used for high-speed, temporary data storage and retrieval, contributing to efficient data processing and control. Sometimes, the DUV will use SRAM to temporarily store a series of inputs for later use. The block diagram example is depicted on the left side of Fig. 3.9.
- **The DUV contains DRAM**: DRAM in an RTL design is used for large, cost-effective data storage and buffering, supporting various applications that require extensive memory capacity and temporary data storage. Sometimes, the DUV will interact with the memory to frequently write and read data. The block diagram example is depicted on the right side of Fig. 3.9.


doi:10.6342/NTU202403664

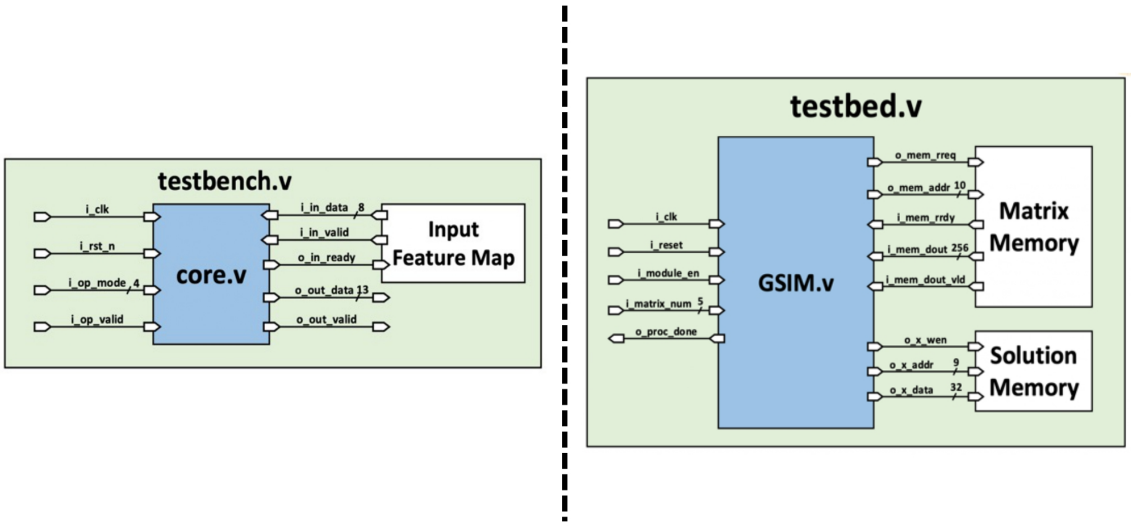



Figure 3.9: The block diagram example of the DUV containing SRAM (left) or DRAM (right)



doi:10.6342/NTU202403664

# Chapter 4 Preparation Phase : Configuration File Setting

In this chapter, we will delve into the detailed parameter information required for the configuration file, a crucial component for setting up our tool. We will cover each parameter in depth to ensure that readers understand their significance and how to properly configure them (Section 4.1). Additionally, we will guide the reader through the process of setting the configuration file using our user interface, making the setup process intuitive and straightforward (Section 4.2). By the end of this chapter, we expect that the reader will have a comprehensive understanding of the configuration file parameters and the steps necessary to configure them correctly.

## 4.1 Configuration File Parameters

In this section, we will introduce the 27 parameters in the configuration file, categorized into five groups. Some parameters are mandatory, such as the path to the DUV and the clock signal name, while others are optional, depending on the properties of the DUV, such as included RTL files and constraints for input signals. The introduction will proceed with an overview of the five categories, and examples of these settings can be found in


doi:10.6342/NTU202403664

Chapter 8.

1. **Design information**:

   - ***-duv***: (mandatory) Designers should provide either the absolute path or a relative path (relative to the root directory of our tool) to specify the location of the DUV.
   - ***-top***: (mandatory) In the verification process, we often employ a divide-and-conquer technique by verifying submodules individually instead of verifying the entire design starting from the top module. Therefore, designers should specify the particular module they intend to perform bug localization on and provide this information in the corresponding parameter. Our tool will treat this module as the pseudo top module.
   - ***-clk***: (mandatory) In a sequential design, the clock signal is essential as it governs the behavior and validity of other signals. Designers need to specify the clock signal name here, since the clock signal in different DUVs may not follow common naming conventions such as “clk” or “i_clk”.
   - ***-rst / -rst_n***: (mandatory) In a sequential design, a reset signal may be used to initialize all internal or output values. This reset signal can be either positive or negative edge-triggered, activating on the rising or falling edge of the clock signal, respectively, and corresponds to *-rst* or *-rst_n* (“_n” standing for “negative”). However, during verification, we typically distinguish the reset signal from other functional verifications. Therefore, we require designers to specify the reset signal name so our algorithm can appropriately ignore it.
   - ***-parameter***: (optional) In the DUV, some constants might be predefined us-


doi:10.6342/NTU202403664

ing the keyword *define* (e.g. define INT_WIDTH 8) or declared directly in the module parameter list (e.g. module top #(INT_WIDTH = 8)). If any parameters are used in control paths (e.g. always, if, case statements), designers need to list them here so that our bug localization algorithm can ignore them.

- ***-rtl_file_list***: (optional) For the DUV, there might be additional RTL files, such as submodules or memory files. However, the third-party tool Pyverilog requires that the input file be flattened, meaning other RTL files cannot be included directly in the DUV. Therefore, designers need to compile a file list that includes all these additional files.
- ***-ref_model***: (mandatory) In our tool, we need a spec to determine whether the DUV's behavior is correct. Therefore, designers must provide a reference model to serve as the spec. To ensure generality, designers should supply a script file in this parameter to execute their reference model. The command and arguments should follow the format: *./<script_file> <input_pattern> <output_result>*.

2. **Simulation information**:

- ***-sim_dir***: (mandatory) The third-party tool SDM matches the output signals between the DUV and the reference model based on the simulation results' waveforms. As introduced in Chapter 2, SDM requires the files "duv_pass.vcd", "duv_fail.vcd", "golden_pass.csv", and "golden_fail.csv", generated from the DUV passing and failing patterns, respectively. Designers should place these four files in a directory and specify that directory in this parameter.
- ***-adapter***: (optional) The concept of this parameter is similar to an adapter in UVM. Since we cannot ensure that designers' input pattern formats for the


doi:10.6342/NTU202403664

DUV and the reference model, or the output result format from the reference model, will always match our tool's format (introduced in Appendix A and B), designers should provide a converter. This converter will translate their formats to our tool's format and vice versa. Specifically, we need an adapter that converts from our tool's input pattern format to their input pattern format and from their output format to our tool's output format. This allows us to generate input patterns, execute them on their reference model, and retrieve the results for our bug localization process.

3. **Supplemental information to enhance bug localization**:

   - ***-verification_assertion***: (optional) While our tool can automatically generate numerous assertions as candidates for bug localization, designers can enhance the results by providing their own verification assertions. Since designers have deeper knowledge of their own designs, their assertions can improve the accuracy of bug localization. These provided assertions will also be filtered through our process and used in the bug localization phase. Note that our algorithm requires these assertions to be written in SystemVerilog Assertions (SVA) and to be concurrent assertions with implications.

   - ***-harm_template***: (optional) As discussed in Chapter 2, the assertions generated by HARM follow a predefined assertion mining template (the details will be introduced in Chapter 5). While our tool employs well-tuned parameters for automatic assertion mining, designers, possessing deeper insights into the DUV, can enhance this process. They have the flexibility to introduce additional LTL templates, propositions, or ranking metrics to guide HARM in generating more contextually relevant assertions for the DUV.


doi:10.6342/NTU202403664

- ***-duv_to_ref_model_output_name_map***: (optional) This parameter addresses a limitation in our tool. Occasionally, the third-party tool SDM exhibits instability in matching output signals, resulting in mismatches where certain DUV output signals may erroneously correspond to reference model output signals. Therefore, until the SDM algorithm is enhanced, if designers encounter errors such as segmentation faults in our tool, it is likely due to incorrect output signal matching. Designers can mitigate this issue by specifying the mapping between the names of output signals in the DUV and their counterparts in the reference model.
- ***-testbench***: (optional) Our tool can automatically generate a testbench for the DUV using the information provided in the configuration file. However, this automation does not apply when the DUV involves DRAM due to the complexity of interactions, which cannot be standardized for testbench generation. Therefore, designers are required to directly provide their own testbench to facilitate the bug localization process.

4. **Input valid signal**:

- ***-ivalid_signal***: (optional) If the DUV has input valid signals indicating the readiness of input signals (Fig.3.7), designers need to list these in this parameter to enable automatic testbench generation in our tool.
- ***-ivalid_phase***: (optional) If the DUV has input valid signals specified in *-ivalid_signal*, designers must also indicate the value or phase at which these signals are activated, usually 1. This typically means that when the input valid signals are high, the input signals are ready.
- ***-ivalid_controlled_signal***: (optional) Sometimes, there may be multiple input

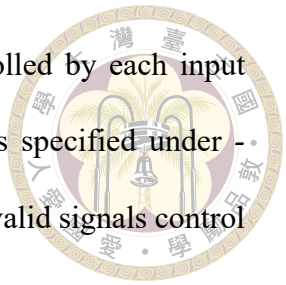

doi:10.6342/NTU202403664

valid signals, each controlling different input signals. Therefore, it is necessary to map each input valid signal to its respective controlled input signals. This mapping enables our tool to precisely control the input data in the automatically generated testbench.

- ***-input_signal_not_in_input_pattern***: (optional) In our process, we generate constrained random patterns to simulate the DUV, excluding the clock signal, reset signal, and input valid signal, as these are used in the testbench to control the simulation. However, as shown in Fig. 4.1 (a), there may be additional input signals that neither control the input data (input valid signal) nor constitute the input data (input signal). These signals, such as *i_module_en*, *i_matrix_num*, and *i_mem_rrdy* in the block diagram, indicate module or memory readiness or act as counters. They are used in the testbench and should not be part of the input patterns. Therefore, designers need to specify these signals to avoid generating random patterns for them.

- ***-input_constraint / -input_constraint_for_multi_ivalid***: (optional) As shown in Fig. 4.1 (b), input signals sometimes need to follow constraints specified in the design specification rather than being purely random in our flow. For instance, the value of the four-bit signal "i_op_mode" Fig. 4.1 (b) should be in the range from 0 to 8. To address this, designers should follow our provided template, as depicted in Fig. 4.1 (c), to create an input constraint file written in SVA format. The template is straightforward: declare a class to write constraints inside and initialize the class in the top module. This information allows us to use the third-party tool VCS for constrained random simulation. The difference between these two parameters is that if there


doi:10.6342/NTU202403664



are multiple input valid signals, the input signals controlled by each input valid signal should have their own input constraint files specified under *-input_constraint_for_multi_ivalid*. This is because input valid signals control their respective input signals independently.

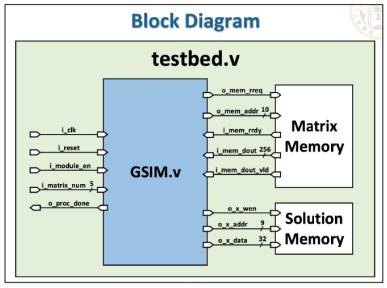
- ***-ivalid_activate_constraint***: (optional) As shown in Fig. 4.1 (d), sometimes the input valid signals must adhere to constraints specified in the design specification, such as certain signals not being high simultaneously. Therefore, designers need to provide these constraints in SVA format. These assertions should be written as concurrent assertions with implications, similar to those specified in the *-verification_assertion* parameter.

- ***-read_in_cycle / -read_in_cycle_for_multi_ivalid***: (mandatory) To automatically generate the testbench, designers must specify the number of cycles an input valid signal needs to remain high to input data. In some DUVs, there may be multiple input valid signals, each functioning independently. For example, in the last figure of Fig. 3.7, the input signal *i_op_mode*, controlled by *i_op_valid*, lasts for one cycle to read an operation, whereas the input signal *i_in_data*, controlled by *i_in_valid*, lasts for 2048 cycles to read all pixels in an image. Therefore, if the DUV has multiple input valid signals, designers should specify the read-in cycle for each under the parameter *-read_in_cycle_for_multi_valid*.

- ***-pattern_num_in_one_sim_round***: (optional, the default value is 1) Some input signals require several patterns to form a meaningful action, which we refer to as “pattern number in one simulation round”. For example, in Fig. 4.1 (b), the input signal *i_op_mode* determines the DUV’s operation, and the spec


doi:10.6342/NTU202403664

mandates that the first operation must be “load image (4’b0000)”. If we simulate this input signal with just one random pattern under the given constraint, the simulation would repeatedly load the image and then terminate, which is unproductive. Therefore, we can set this parameter to 100, indicating that after loading the image, 99 additional random operations can be performed in this simulation iteration. These 100 patterns collectively represent a “meaningful action”.




(a)

| i_op_mode | Meaning |
|---|---|
| 4’b0000 | Input feature map loading |
| 4’b0001 | Origin right shift |
| 4’b0010 | Origin left shift |
| 4’b0011 | Origin up shift |
| 4’b0100 | Origin down shift |
| 4’b0101 | Reduce the number of channels for convolution |
| 4’b0110 | Increase the number of channels for convolution |
| 4’b0111 | Perform convolution in the display region |
| 4’b1000 | Output the pixels in the display region |

(b)

```
class Spec;
    rand bit [3:0] mode;
    rand bit       en;
    static int counter = 0;
    constraint c_mode {
        mode > 2;
        mode <= 6;
    }

endclass

module tb;
    Spec spec;
    initial begin
        spec = new();
        for (int i = 0 ; i < 1 ; i++) begin
            spec.randomize();
            $display("[VCS input constraint] mode = %d", spec.mode);
            $display("[VCS input constraint] en = %d", spec.en);
            ++spec.counter;
        end
        $finish;
    end
endmodule
```

(c)

7. i_op_valid stays only **1 cycle**.
8. i_in_valid and o_op_ready can’t be **high** in the same time.
9. i_op_valid and o_op_ready can’t be **high** in the same time.
10. i_in_valid and o_out_valid can’t be **high** in the same time.
11. i_op_valid and o_out_valid can’t be **high** in the same time.
12. o_op_ready and o_out_valid can’t be **high** in the same time.

(d)

Figure 4.1: The examples illustrating the configuration file parameters

5. **Output valid signal**:

- ***-ovalid_signal***: (optional) If the DUV has output valid signals indicating the readiness of output signals (Fig.3.8), designers need to list these in this parameter to enable automatic testbench generation in our tool.
- ***-ovalid_phase***: (optional) If the DUV has output valid signals specified in *-ovalid_signal*, designers must also indicate the value or phase at which these signals are activated, usually 1. This typically means that when the output


doi:10.6342/NTU202403664

valid signals are high, the output signals are ready.

- ***-ovalid_controlled_signal***: (optional) Sometimes, there may be multiple output valid signals, each controlling different output signals. Therefore, it is necessary to map each output valid signal to its respective controlled output signals. This mapping allows our tool to accurately retrieve and compare the correct output pairs between the DUV and the reference model.

## 4.2 Setting the Configuration File through User Interface

Designers can configure the settings manually by modifying the template according to our documentation, or they can set the *-guide_for_config_file* flag in the command to activate the user interface, which guides them through the configuration file setup step-by-step. In this section, we will present our user interface for the configuration file setting.

As depicted in Fig. 4.2, when the user interface is activated, a greeting message *Starting guidance...* appears, signaling the start of the process. At each step, a progress bar informs designers of the remaining parameters to be set. Below the progress bar is the current parameter to be configured. For example, in Fig. 4.2, the parameter *-duv* needs to be set and will be stored as a string type in our configuration file. A description provides the parameter's meaning, and the number of arguments that designers need to provide will be indicated in the *amount* section. Additionally, an example is provided to avoid ambiguity. Finally, we specify the state of the parameter: some parameters are mandatory (e.g. path to the DUV, clock signal name), while others are optional (e.g. included RTL files, constraints for input signals). Once all parameters are set, the configuration file is automatically generated, and our bug localization tool begins its process.


doi:10.6342/NTU202403664

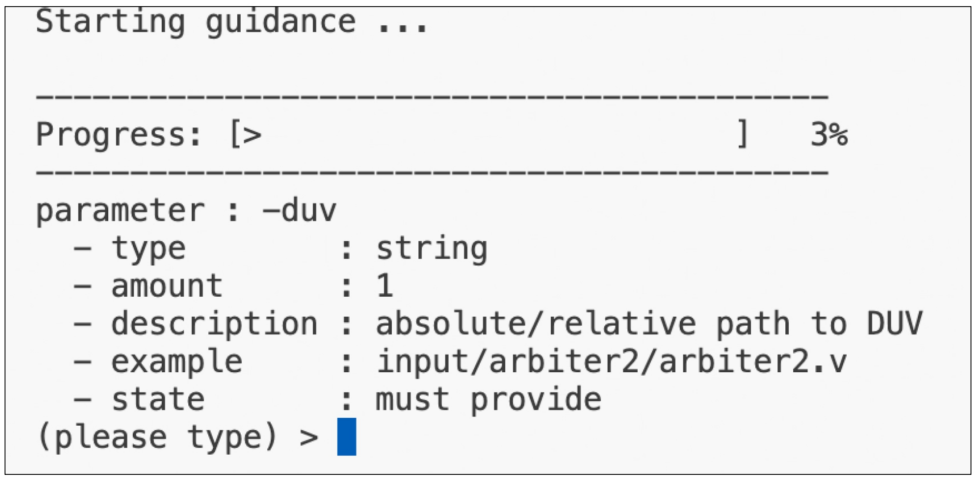



Figure 4.2: The demonstration of the configuration file user interface (start setting)

Note that for parameters marked as must-provide, the interface will prevent designers from proceeding if no input is given (e.g. pressing enter without typing anything), as shown in Fig. 4.3. Once the current setting is provided, we can move on to the next parameter.

In addition to the *string* and *int* types, our settings also include *vector* and *unordered_map* types. Simply follow the guidance and parameter examples provided. For instance, in Fig. 4.4, the *-parameter* will store an array of strings, allowing designers to input each defined variable in the DUV one by one. Similarly, the *ivalid_controlled_signal* will store mappings between input valid signals and their controlled signals; designers can input each mapping pair sequentially until all are listed.

In summary, once the configuration file settings are complete, all necessary information for the subsequent phases is finalized. We can then proceed to the next phase.


doi:10.6342/NTU202403664

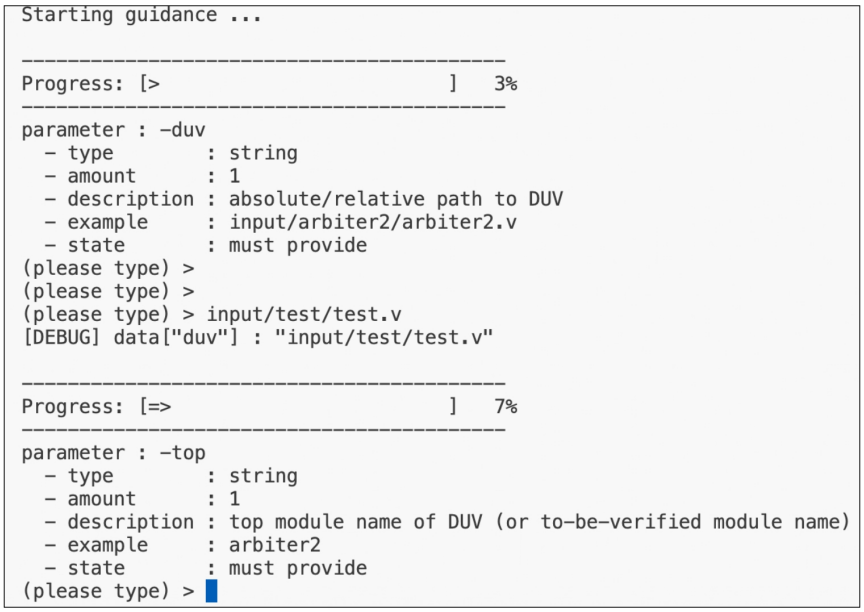



Figure 4.3: The demonstration of the configuration file user interface (successfully set for string or int type)

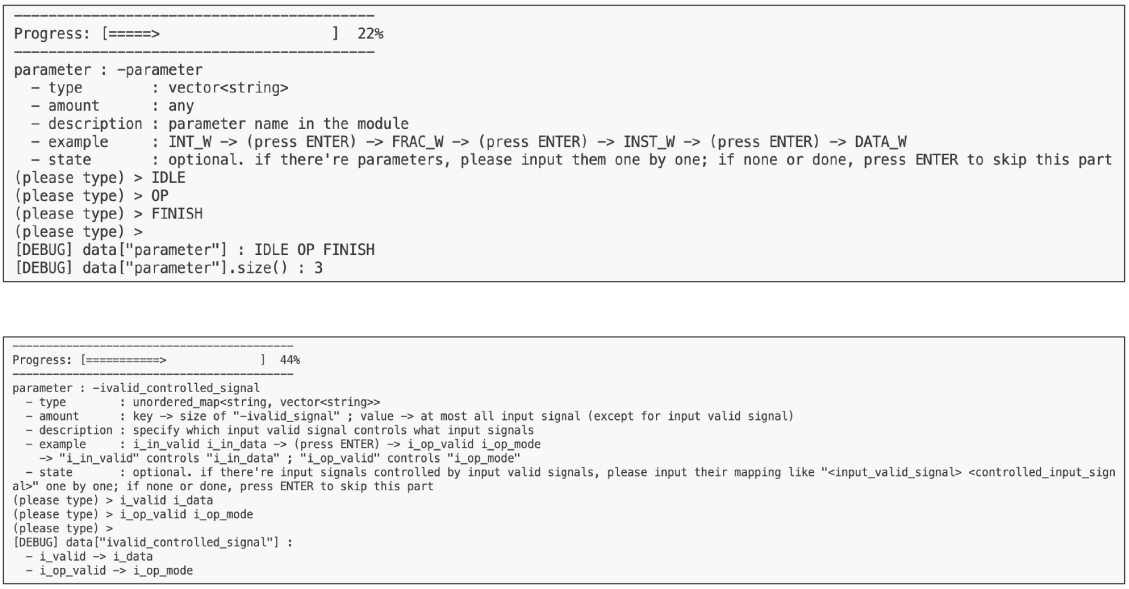



Figure 4.4: The demonstration of the configuration file user interface (successfully set for array type and map type)



doi:10.6342/NTU202403664

# Chapter 5 Assertion Generation Phase : Valid Assertion Mining from the DUV

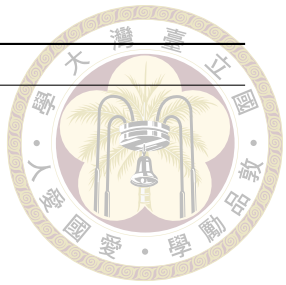
In this chapter, we delve into the algorithmic process of generating valid assertions for the DUV. The following sections provide a detailed exploration of the methods and tools used. In section 5.1, we present a broad overview of the assertion mining algorithm. The subsequent sections describe each component of our framework in detail, along with the corresponding algorithms. These sections collectively aim to provide a comprehensive understanding of how valid assertions are generated and verified.

## 5.1 Overview of the Assertion Mining Algorithm

After configuring the file in Chapter 4, we use the information provided by the designers to generate valid assertions, as defined in Chapter 3. In Algorithm 1, we will divide into three parts to introduce: preprocessing the DUV (Section 5.1), generating assertions using HARM (Section 5.2), and filtering valid assertions as candidates with VC Formal (Section 5.3).

First, we use the third-party tool SDM to match output signals between the DUV and the reference model (line 1). Using the simulation result where the DUV passes


doi:10.6342/NTU202403664



**Algorithm 1** Valid Assertion Mining Algorithm

**Input:** Configuration file
**Output:** Valid assertions
1: *SDM_Output_Matching()*
2: *Classify_Signal_Type()*
3: *Extract_Cared_Signal()*
4: *HARM_Tcl_Generation(genTemplate=True)*
5: *Modify_HARM_Template()*
6: *HARM_Tcl_Generation(genTemplate=False)*
7: *Parse_and_Patch_Assertion_to_SVA()*
8: *VCF_Tcl_Generation()*
9: *Filter_Valid_Assertion()*

(*duv_pass.vcd*) and the golden result of the reference model under the same input patterns (*golden_pass.vcd*) from the configuration file, SDM's main algorithm [33] converts the output signal values from *duv_pass.vcd* and *golden_pass.vcd* to binary strings, as shown in Fig. 5.1 (a). This output value matching problem is then treated as an approximate string matching problem (Fig. 5.1 (b)), utilizing algorithms like the Myers difference algorithm to attempt matching. Considering the correspondence between signals, SDM accounts for both one-to-one mappings and one-to-many relationships from the reference model to the DUV. This is modeled as a set cover problem as depicted in Fig. 5.1 (c), with a greedy approach employed to find the best match. Finally, the output signal matching from the DUV to the reference model is obtained automatically after applying the "*SDM_Output_Matching()*" function.

Since it is impractical to ask designers to provide all input and output signal names, our tool directly parses the DUV's I/O port declaration to extract these names (line 2). However, merely extracting this information is insufficient; we also need to filter out unrelated signals, typically most of the DUV's internal signals. Therefore, we perform the "*Extract_Cared_Signal()*" function (line 3) to exclude the clock signal, reset signal, defined parameters, and internal signals not present in the control path (i.e., always, if/case


doi:10.6342/NTU202403664

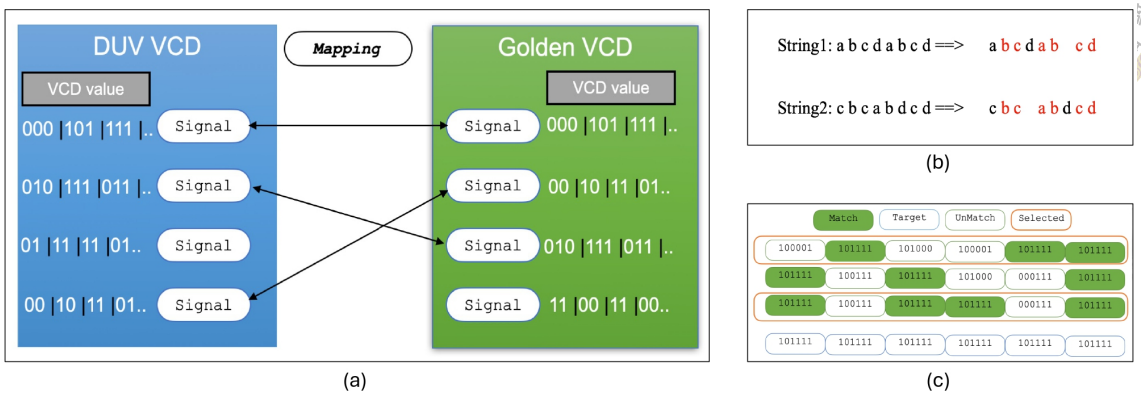



Figure 5.1: The illustration of concepts in SDM' s algorithm includes: (a) sequential matching, (b) approximate string matching, and (c) sequential matching in the set cover problem

statements). This process retains only the input signals, output signals, and control path signals for further assertion generation by HARM, focusing on behavior crucial to the bug localization process.

## 5.2 Assertion Mining by HARM

After preprocessing the signals, we can generate assertions for the DUV using HARM. HARM provides an option to automatically generate a basic template (line 4) by setting the flag "*–generate-config*" in the command, as shown in Fig. 5.2 (a). Additionally, we need the failing waveform "*duv_fail.vcd*", the clock signal from the configuration file, and the module scope parsed from the VCD file by our tool. This information is used to automatically generate a basic template for HARM, as depicted in Fig. 5.2 (b), guiding HARM to mine assertions.


doi:10.6342/NTU202403664


(a)

```
1   <harm>
2       <context name="default">
3
4           <prop exp="state" loc="c,dt"/>
5           <prop exp="gnt1" loc="c,dt"/>
6           <prop exp="rst" loc="c,dt"/>
7           <prop exp="gnt2" loc="c,dt"/>
8           <prop exp="req2" loc="c,dt"/>
9           <prop exp="req1" loc="c,dt"/>
10          <prop exp="clk" loc="c,dt"/>
11
12          <template dtLimits="5D,3W,15A,-0.1E,R" exp="G({..#1&..}|-> P0)" />
13
14          <sort name="causality" exp="1-afct/traceLength"/>
15          <sort name="frequency" exp="atct/traceLength"/>
16      </context>
17  </harm>
```

(b)

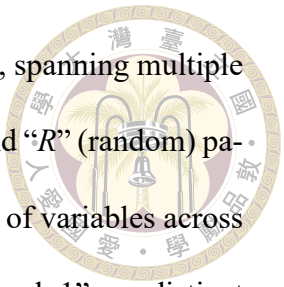
Figure 5.2: The example of HARM's template includes: (a) the command used to automatically generate the template, and (b) the basic template produced by the aforementioned command

In Fig. 5.3 (a), we see that the basic template for HARM consists of three main components: propositions (beginning with "*prop*"), LTL templates (beginning with "*template*"), and ranking metrics (beginning with "*sort*"). Propositions are used to form assertions by permuting with other propositions. In the basic template shown in Fig. 5.3 (a), HARM considers all types of signals (input, output, internal) as proposition candidates, which can be placed at "*c*" (consequent of an assertion) and "*dt*" (decision tree operator to be the state transition's variable).

In our modification (line 5), we focus only on the "cared signals" extracted in line 3. This reduces the number of assertion candidates, thereby accelerating the process and concentrating on signals crucial to the control path. The modified template is shown in Fig. 5.3 (b).

For the LTL template, placeholders are used to fill in propositions. To generate diverse assertions, we include various placeholders such as "...*#1&*...", "...*##1*...", and "...


doi:10.6342/NTU202403664



&&...” to cover assertions with and without temporal dimensions (i.e., spanning multiple cycles). For assertions crossing cycles, we use both “*S*” (sequential) and “*R*” (random) parameters to generate assertions that either consider or ignore the order of variables across cycles. For example, “signal_1 ##1 signal_2” and “signal_2 ##1 signal_1” are distinct under parameter “*S*” but equivalent under parameter “*R*”.

Other parameters, introduced in Chapter 2, are left untuned as they already provide effective assertion mining results. Empirically, deeper temporal dimensions (parameter “*D*”), more propositions (parameters “*W*” and “*A*”), and more decision tree candidates (parameter “*E*”) do not significantly alter the mined assertions. We also do not modify the ranking metrics, which sort the generated assertions based on their scores.

These modifications make our tool more general in generating assertions while focusing on possible bug behaviors.

```
<harm>
    <context name="default">

        <prop exp="state" loc="c,dt"/>
        <prop exp="gnt1" loc="c,dt"/>
        <prop exp="rst" loc="c,dt"/>
        <prop exp="gnt2" loc="c,dt"/>
        <prop exp="req2" loc="c,dt"/>
        <prop exp="req1" loc="c,dt"/>
        <prop exp="clk" loc="c,dt"/>

        <template dtLimits="5D,3W,15A,~0.1E,R" exp="G({..#1&..}|-> P0)" />

        <sort name="causality" exp="1-afct/traceLength"/>
        <sort name="frequency" exp="atct/traceLength"/>
    </context>
</harm>
```

(a)

```
<harm>
    <context name="default">

        <prop exp="state" loc="a,c,dt"/>
        <prop exp="gnt1" loc="a,c,dt"/>
        <prop exp="gnt2" loc="a,c,dt"/>
        <prop exp="req2" loc="a,c,dt"/>
        <prop exp="req1" loc="a,c,dt"/>

        <template dtLimits="5D,3W,15A,~0.1E,R" exp="G({..&&..}|-> P0)" />
        <template dtLimits="5D,3W,15A,~0.1E,S" exp="G({..&&..}|-> P0)" />
        <template dtLimits="5D,3W,15A,~0.1E,R" exp="G({..##1..}|-> P0)" />
        <template dtLimits="5D,3W,15A,~0.1E,S" exp="G({..##1..}|-> P0)" />
        <template dtLimits="5D,3W,15A,~0.1E,R" exp="G({..#1&..}|-> P0)" />
        <template dtLimits="5D,3W,15A,~0.1E,S" exp="G({..#1&..}|-> P0)" />

        <sort name="causality" exp="1-afct/traceLength"/>
        <sort name="frequency" exp="atct/traceLength"/>
    </context>
</harm>
```

(b)

Figure 5.3: The comparison of the template before and after applying the “*Modify_HARM_Template()*” function includes: (a) the original basic template generated by HARM, and (b) the modified template produced by our tool

At the end of the assertion mining process, we will run the command shown in Fig. 5.2



doi:10.6342/NTU202403664

(a) again, but this time without the "*–generate-config*" flag (line 6). This initiates automatic assertion generation based on our modified template in Fig. 5.2 (b). After executing HARM, we obtain the assertion log results, as shown in Fig. 5.4. These assertions are based on the modified template's propositions (considering only cared signals) and LTL templates ("...*#1&*...", "...*##1*...", and "...*&&*..."). They are sorted by scores calculated according to the ranking metrics in the template.

Next, we will convert the assertion log into SVA format and perform formal verification on these assertions and the DUV to filter valid assertions.

```
123  [INFO] 03:27:35 - Message: Metrics filtered 0 assertions
124
125  | N  | Assertion (Context : default)                                                                  | final | causality | frequency |
126  |----|------------------------------------------------------------------------------------------------|-------|-----------|-----------|
127  | 0  | G({(req1)} |-> gnt1)                                                                           | 1.00  | 1.00      | 1.00      |
128  | 1  | G({(gnt1)} |-> req1)                                                                           | 1.00  | 1.00      | 1.00      |
129  | 2  | G({(req1) ##1 (true)} |-> state)                                                                | 0.99  | 0.74      | 1.00      |
130  | 3  | G({(gnt1) ##1 (true)} |-> state)                                                                | 0.99  | 0.74      | 1.00      |
131  | 4  | G({(gnt2)} |-> req2)                                                                           | 0.97  | 0.86      | 0.65      |
132  | 5  | G({(req2 && state)} |-> gnt2)                                                                   | 0.42  | 0.89      | 0.40      |
133  | 6  | G({(gnt2 && req1)} |-> state)                                                                   | 0.00  | 0.63      | 0.14      |
134  | 7  | G({(gnt1 && gnt2)} |-> state)                                                                   | 0.00  | 0.63      | 0.14      |
135  | 8  | G({(gnt2) ##2 (state) ##1 (req2) ##1 (true)} |-> req1)                                           | 0.00  | 0.51      | 0.09      |
136  | 9  | G({(gnt2) ##2 (state) ##1 (req2) ##1 (true)} |-> gnt1)                                           | 0.00  | 0.51      | 0.09      |
137  | 10 | G({state ##2 req2 ##2 gnt2} |-> state)                                                          | 0.00  | 0.87      | 0.07      |
138  | 11 | G({state ##1 req2 ##2 gnt2} |-> state)                                                          | 0.00  | 0.83      | 0.07      |
139  | 12 | G({(state) ##1 (state) ##2 (gnt2) ##1 (true)} |-> req2)                                          | 0.00  | 0.63      | 0.07      |
140  | 13 | G({gnt1 ##2 req2 ##2 gnt2} |-> state)                                                          | 0.00  | 0.57      | 0.07      |
141  | 14 | G({req1 ##2 req2 ##2 gnt2} |-> state)                                                          | 0.00  | 0.57      | 0.07      |
142  | 15 | G({state ##2 state ##1 gnt2 ##1 !gnt1} |-> state)                                               | 0.00  | 0.95      | 0.02      |
143  | 16 | G({state ##2 state ##1 gnt2 ##1 !req1} |-> state)                                               | 0.00  | 0.95      | 0.02      |
144  | 17 | G({gnt2 ##1 state ##2 gnt2 ##1 true} |-> gnt2)                                                  | 0.00  | 0.94      | 0.02      |
145  | 18 | G({gnt2 ##1 state ##2 state ##1 true} |-> gnt2)                                                | 0.00  | 0.94      | 0.02      |
146  | 19 | G({gnt2 ##1 state ##3 req2} |-> gnt2)                                                          | 0.00  | 0.94      | 0.02      |
147  | 20 | G({gnt2 ##1 state ##2 req2 ##1 true} |-> gnt2)                                                | 0.00  | 0.94      | 0.02      |
148  | 21 | G({gnt2 ##1 state ##1 gnt1 ##2 true} |-> gnt2)                                                | 0.00  | 0.92      | 0.02      |
149  | 22 | G({gnt2 ##1 state ##1 req1 ##2 true} |-> gnt2)                                                | 0.00  | 0.92      | 0.02      |
150  | 23 | G({(state) ##2 (!gnt2 && state) ##1 (gnt2 && state) ##1 (true)} |-> state)                        | 0.00  | 0.91      | 0.02      |
151  | 24 | G({(state) ##2 (req2) ##1 (gnt2 && req1) ##1 (true)} |-> gnt2)                                  | 0.00  | 0.85      | 0.02      |
152  | 25 | G({(req1) ##3 (!gnt2 && gnt1 && state) ##1 (gnt2)} |-> req1)                                    | 0.00  | 0.84      | 0.02      |
153  | 26 | G({(req1) ##3 (!gnt2 && gnt1 && state) ##1 (gnt2)} |-> gnt1)                                    | 0.00  | 0.84      | 0.02      |
154  | 27 | G({(gnt1) ##3 (!gnt2 && gnt1 && state) ##1 (gnt2)} |-> req1)                                    | 0.00  | 0.84      | 0.02      |
155  | 28 | G({(state) ##2 (!gnt2 && gnt1 && state) ##1 (gnt2) ##1 (true)} |-> state)                          | 0.00  | 0.84      | 0.02      |
156  | 29 | G({(gnt1) ##3 (!gnt2 && gnt1 && state) ##1 (gnt2)} |-> gnt1)                                    | 0.00  | 0.84      | 0.02      |
```

Figure 5.4: The example of the assertion log file generated by HARM

## 5.3 Formal Verifying Assertions by VC Formal

After generating numerous assertions using HARM, we must verify their validity through formal verification using the third-party tool VC Formal. To start, we need to prepare the DUV and the mined assertions in a compatible format. We do this by calling the function "*Parse_and_Patch_Assertion_to_SVA()*" (line 7), which parses the assertions


doi:10.6342/NTU202403664

from the assertion log file shown in Fig. 5.4. These parsed assertions are then wrapped in SVA format and appended to the end of the module to be verified (i.e. the top module specified in the configuration file), as illustrated in Fig. 5.5.

```
1  module arbiter2(clk, rst, req1, req2, gnt1, gnt2);
2      input clk, rst;
3      input req1, req2;
4      output gnt1, gnt2;
5      reg state;
6      reg gnt1, gnt2;
7      always @(posedge clk or posedge rst) begin
8          if (rst) state <= 0;
9          else state <= gnt1;
10     end
11     always @(*) begin
12         if (state) begin
13             gnt1 = req1; // bug
14             // gnt1 = req1 & ~req2; // golden
15             gnt2 = req2;
16         end
17         else begin
18             gnt1 = req1;
19             gnt2 = req2 & ~req1;
20         end
21     end
22
23     // mined assertion
24     `define true 1
25     assert_mined_0: assert property(@(posedge clk) ((req1)) |-> gnt1);
26     assert_mined_1: assert property(@(posedge clk) ((gnt1)) |-> req1);
27     assert_mined_2: assert property(@(posedge clk) ((req1) ##1 (`true)) |-> state);
28     assert_mined_3: assert property(@(posedge clk) ((gnt1) ##1 (`true)) |-> state);
29     assert_mined_4: assert property(@(posedge clk) ((gnt2)) |-> req2);
30     assert_mined_5: assert property(@(posedge clk) ((req2 && state)) |-> gnt2);
31     assert_mined_6: assert property(@(posedge clk) ((gnt2 && req1)) |-> state);
32     assert_mined_7: assert property(@(posedge clk) ((gnt1 && gnt2)) |-> state);
33     assert_mined_8: assert property(@(posedge clk) ((gnt2) ##2 (state) ##1 (req2) ##1 (`true)) |-> req1);
34     assert_mined_9: assert property(@(posedge clk) ((gnt2) ##2 (state) ##1 (req2) ##1 (`true)) |-> gnt1);
35     assert_mined_10: assert property(@(posedge clk) (state ##2 req2 ##2 gnt2) |-> state);
36     assert_mined_11: assert property(@(posedge clk) (state ##1 req2 ##2 gnt2) |-> state);
37     assert_mined_12: assert property(@(posedge clk) ((state) ##1 (state) ##2 (gnt2) ##1 (`true)) |-> req2);
```

Figure 5.5: The example of the DUV with patched assertions

Next, we will automatically set up the script file based on the design information provided by the designers and the DUV with patched assertions to execute VC Formal (line 8), as shown in Fig. 5.6. We will use the command "*check_fv*" to perform formal verification and "*report_fv*" to generate the verification report. This report will indicate whether each mined assertion is proven, falsified, or vacuous with respect to the DUV.

Finally, we will obtain the formal verification report from VC Formal, as illustrated in Fig. 5.7. This report details the verification status of each mined assertion, indicating whether it is proven, falsified (with the specific depths or transitions required to detect the counterexample), or vacuous (no antecedent signal is triggered). As discussed in Section 3.3, we will retain only the assertions that are proven with respect to the DUV as our valid assertions for the next phase (line 9).


doi:10.6342/NTU202403664

```
set_fml_appmode FPV

#########################################################
## Setup Specific to DUT
#########################################################
set design arbiter2

#########################################################
## Compile & Setup
#########################################################
# Compilation Step
read_file -top $design -format sverilog -sva -vcs {input/arbiter2/temp/arbiter2.sva}

# Initialisation Commands
sim_run -stable
sim_save_reset

#########################################################
## Proof
#########################################################
check_fv -block

#########################################################
## Reports
#########################################################
report_fv -list > input/arbiter2/temp/formal_result.txt
report_fv -verbose > input/arbiter2/temp/formal_result_verbose.txt
quit
```

Figure 5.6: The example of VC Formal setup

```

Summary Results
 Property Summary: FPV
 ------------------
 > Assertion
   - # found        : 93
   - # proven       : 6
   - # falsified    : 87

 > Vacuity
   - # found        : 93
   - # non_vacuous  : 93


List Results
 Property List:
 --------------
 > Assertion
   # Assertion: 93
   [   0] proven                  (non_vacuous)  -  arbiter2.assert_mined_0
   [   1] proven                  (non_vacuous)  -  arbiter2.assert_mined_1
   [   2] falsified   (depth=12)  (non_vacuous)  -  arbiter2.assert_mined_10
   [   3] falsified   (depth=10)  (non_vacuous)  -  arbiter2.assert_mined_11
   [   4] falsified   (depth=26)  (non_vacuous)  -  arbiter2.assert_mined_12
   [   5] falsified   (depth=12)  (non_vacuous)  -  arbiter2.assert_mined_13
   [   6] falsified   (depth=12)  (non_vacuous)  -  arbiter2.assert_mined_14
   [   7] falsified   (depth=8)   (non_vacuous)  -  arbiter2.assert_mined_15
   [   8] falsified   (depth=8)   (non_vacuous)  -  arbiter2.assert_mined_16
   [   9] falsified   (depth=30)  (non_vacuous)  -  arbiter2.assert_mined_17
```

Figure 5.7: The example of VC Formal report



doi:10.6342/NTU202403664

Note that if designers provide their verification assertions (assumed to be golden) via the "-*verification_assertion*" parameter discussed in Section 4.1, we will prioritize assertions in the "falsified" state from VC Formal as our valid assertion candidates for bug localization. A "falsified" assertion indicates that the DUV does not satisfy the golden spec, necessitating further investigation into the assertion, as illustrated on the left side of Fig. 5.8. This makes it a strong candidate for bug localization. Conversely, a "proven" assertion merely indicates that the assertion's state space includes both the DUV and the reference model, as shown on the right side of Fig. 5.8. When a bug is triggered, there's no reason to question a "proven" assertion for bug localization, so it will not be used as a candidate.

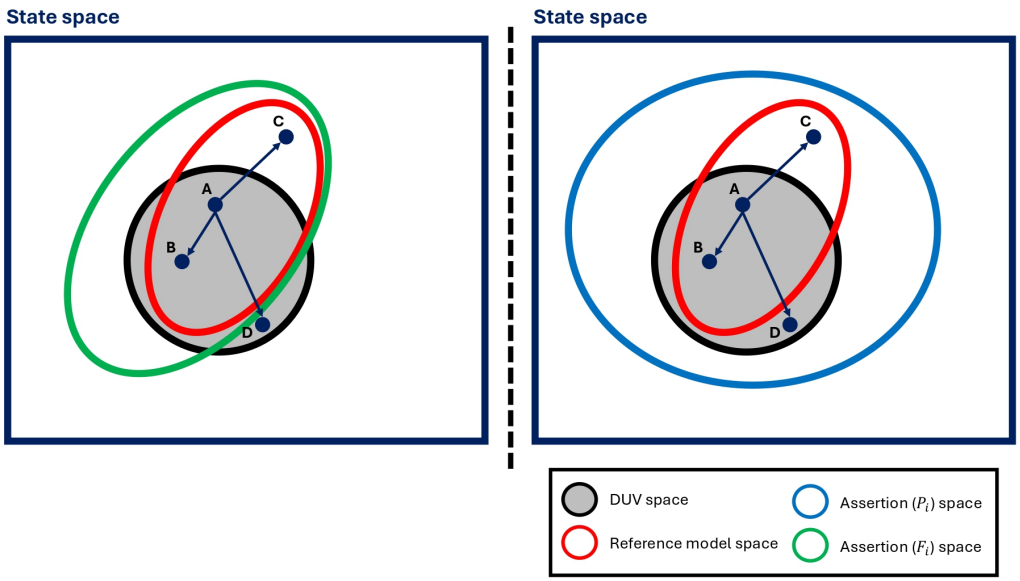



Figure 5.8: The illustration of the state space relationship among the golden assertion provided by designers, the reference model, and the DUV

In summary, during this assertion generation phase, HARM generates assertions based on the simulation results provided by designers, encompassing both the blue and green circles (state spaces) in Fig. 3.6. VC Formal then filters these to retain only the blue circle (state space) assertions, which will be used as candidates in the next phase.


doi:10.6342/NTU202403664

# Chapter 6 Assertion Validation Phase : Suspected Assertion Screening

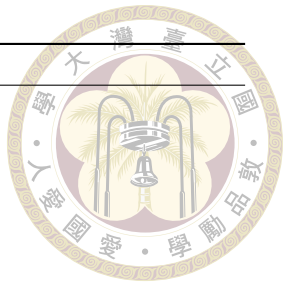
In this chapter, we aim to validate the assertions generated in the previous phase and identify those that are suspected for bug localization. The process involves constrained random simulation to ensure that only relevant assertions are considered as final candidates. The following sections will provide a comprehensive exploration of the methods and algorithms employed in this phase.

## 6.1 Overview of the Suspected Assertion Screening Algorithm

This section provides a high-level overview of the algorithm designed to filter suspected assertions. We will introduce three functions in separate sections, as their underlying algorithms require detailed explanation. These sections, from Section 6.2 to Section 6.4, will cover these functions in depth.

In Alg. 2, we present an overview of the Assertion Validation Phase. Starting with the valid assertion candidates generated from Alg. 1, our goal in this phase is to classify these assertions into three categories: golden, failed, and invalid with respect to the reference


doi:10.6342/NTU202403664



**Algorithm 2** Suspected Assertion Screening Algorithm

**Input:** $validAssertions$, configuration File
**Output:** Golden / Failed / Invalid assertions

1: **for each** $\mathbb{A}$ in $validAssertions$ **do**
2: **if** only internal signals in $\mathbb{A}$ **then**
3: Mark $\mathbb{A}$ as $invalid$
4: **continue**
5: $inputPattern \leftarrow$ *Generate_Assertion_Pattern*($\mathbb{A}$)
6: $refResult \leftarrow$ *Hit_Pattern_to_Ref_Model*($inputPattern, False$)
7: $duvResult \leftarrow$ *Generate_Testbench_and_Hit_Pattern*($inputPattern$)
8: $state \leftarrow$ *Compare_with_Ref_Model*($\mathbb{A}$, $inputPattern, duvResult, refResult$)
9: **if** $state == golden$ or $failed$ **then**
10: Mark $\mathbb{A}$ as $state$
11: **else if** $state == invalid$ **then**
12: $anotherTryAssertions$.*push_back*($\mathbb{A}$)
13: $refResult \leftarrow$ *Hit_Pattern_to_Ref_Model*($inputPattern, True$)
14: $duvResult \leftarrow$ *Generate_Testbench_and_Hit_Pattern*($inputPattern$)
15: **for each** $\mathbb{A}'$ in $anotherTryAssertions$ **do**
16: $state' \leftarrow$ *Compare_with_Ref_Model*($\mathbb{A}'$, $inputPattern, duvResult, refResult$)
17: Mark $\mathbb{A}'$ as $state'$

model. Assertions in the failed category, which are valid for the DUV but fail against the reference model (representing the spec), will be identified as suspected assertions. These will then be input into the final phase to be mapped back to the RTL code.

First, we will check the validity of each assertion $\mathbb{A}$ individually (lines 2-4). Since the reference model is treated as a black box, we do not impose any constraints on its internal implementation, aside from its input and output ports. Consequently, the DUV and the reference model typically do not share any internal signals. Under the assertion template *assert(<antecedent> |→ <consequent>)*, if an assertion mined from the DUV consists entirely of internal signals in both the antecedent and consequent, we cannot use this assertion to connect with the reference model. This makes the assertion invalid for our bug localization process, and it will be categorized as "invalid".

Next, we will generate constrained random patterns based on the value range of the antecedent signals (line 5). These patterns will be applied to both the reference model


doi:10.6342/NTU202403664

(line 6) and the DUV (line 7). Finally, we will compare the results at the transaction level and extract the waveform of the assertion to determine if it is suspected based on the output result (line 8). The assertions will then be categorized as "golden", "failed", or "invalid" state with respect to the reference model. The detailed processes for these steps are elaborated in Sections 6.2 to 6.4.

For assertions categorized as "invalid", we cannot determine if they are golden or failed under the current input pattern. Therefore, we will give them another try (lines 11-12). These assertions will be collected, and a one-time purely random simulation will be performed using a similar process (lines 13-16). The only difference is that if the assertion remains in the "invalid" state after this additional simulation, it will be definitively classified as "invalid".

Finally, assertions categorized as "failed" after filtering will be passed to the next phase, where they will be used to localize the suspected bug zones in the RTL code for designers.

## 6.2 Assertion Pattern Generation Algorithm

In this section, we will explore the algorithms referenced in lines 5-6 of Alg. 2, specifically Alg. 3 and Alg. 4.

In Alg. 3, given an assertion in the implication format *assert(<antecedent> |→ <consequent>)*, our goal is to generate an input pattern set that maximizes the likelihood of observing waveforms that satisfy the assertion's implication when applied to the DUV. The key concept of this algorithm is to satisfy the range of input signals specified in the antecedent of the assertion and then observe whether the DUV behaves as expected ac-


doi:10.6342/NTU202403664

**Algorithm 3** Generate Assertion Pattern

**Input:** $assertion$
**Output:** $inputPattern$

$assertionDepth \leftarrow$ *Parse_Assertion_Cycle*$(assertion)$
**if** has given input constraint **then** $iter \leftarrow 1$
**else** $iter \leftarrow \max(1, \frac{iter \times 10}{readInCycle})$
$antExpected \leftarrow$ *Store_Antecedent_Expected_Range*$(assertion)$
$consExpected \leftarrow$ *Store_Consequent_Expected_Range*$(assertion)$
**for** $i \leftarrow 1$ to $iter$ **do**
    **if** has given input constraint **then**
        $inputPattern \leftarrow$ *Generate_Pattern_with_User_Input_Constraint()*
    **else**
        $patternLength \leftarrow 0$
        **for each** $sig_{in}$ in $inputSigs$ **do**
            **if** $sig_{in}$ is input valid signal **then**
                **continue**
            **if** $readInCycle == 1$ **then**
                $windowSize \leftarrow assertionDepth$
            **else if** $readInCycle > 1$ **then**
                $windowSize \leftarrow readInCycle$
            **else**
                $windowSize \leftarrow readInCycleMultiIvalid[controlIvalid[sig_{in}]]$
            **for** $cycle \leftarrow 1$ to $windowSize$ **do**
                **if** ($sig_{in}$ not in the antecedent) **or** ($cycle \geq assertionDepth$) **then**
                    $randPat \leftarrow$ *Generate_Pure_Random_Pattern*$(sig_{in}.width)$
                **else**
                    **if** $rangeConflict$ **then**
                        $randPat \leftarrow$ *Generate_Random_Pat_in_Range*$(cycle)$
                    **else**
                        $randPat \leftarrow$ *Generate_Random_Pat_in_Intersected_Range()*
            $inputPattern[sig_{in}]$.*push_back*$(randPat)$
        $patternLength \mathrel{+}= windowSize$


doi:10.6342/NTU202403664

cording to the consequent.

Before delving into the algorithm, we first illustrate the core concept to aid understanding, as shown in Fig. 6.1. The "assertion depth" or "assertion's crossing cycle" mentioned below refers to the number of cycles an assertion spans to represent a complete implication. For example, *assert(a ##1 b |→ c)* has a depth of 2: starting from $a$ at cycle 0, after one cycle (##1), if $b$ occurs, it implies that $c$ should happen in the same cycle, thus crossing 2 cycles. Generally, an assertion can be divided into two scenarios: (1) and (2) in Fig. 6.1.

In scenario (1), the intersection of the value ranges for each antecedent input signal (regardless of whether the signal spans multiple cycles) is not empty. For example, consider input signals $a$ and $b$. The value range of $a$ across all assertion depths can be from 1 to 3. For $b$, the intersected range of "$b == 1$" and "$(b \geq 1)$ and $(b \leq 3)$" is "$b == 1$". As illustrated in the table for case (1), we list the input pattern range that will be randomly generated for input signals $a$ and $b$ across cycles 0 to 4 based on the assertion example.

In scenario (2), there may be a range conflict in the antecedent input signal (assuming $a$ is an input signal). In this case, we will generate a random pattern that satisfies each value range on a cycle-by-cycle basis. For example, as shown in the table for case (2), we will generate a random value for $a$ within the range $a == 1$ in the first cycle, and within the range "$(a \geq 3)$ and $(a \leq 5)$" in the second cycle, and so on.

Note that we initially consider only input signals when generating patterns, as our goal is to increase the probability of producing the assertion waveform in the simulation results. After feeding in the generated input patterns, we will observe the appearance of the assertion waveform and then take all types of signals into account, including input,


doi:10.6342/NTU202403664

output, and internal signals.

Returning to Alg. 3, we first parse the assertion to determine its depth (line 1). Then, we set the iteration count empirically (lines 2-3) to balance between runtime efficiency and result quality. Subsequently, we parse the antecedent and consequent to extract the value ranges for each input signal at every cycle within the assertion depth for the upcoming constrained random pattern generation (lines 4-5).

For each simulation iteration, we first check if designers have provided input constraints. If constraints are available, we use the third-party tool VCS to generate constrained random patterns based on the input constraint file specified in *-input_constraint* in Section 4.1, incorporating our tool's range constraints as soft constraints (lines 7-8). If no input constraint file is provided, we follow the previously explained process with Fig. 6.1, resulting in the $inputPattern$ for the subsequent simulation.

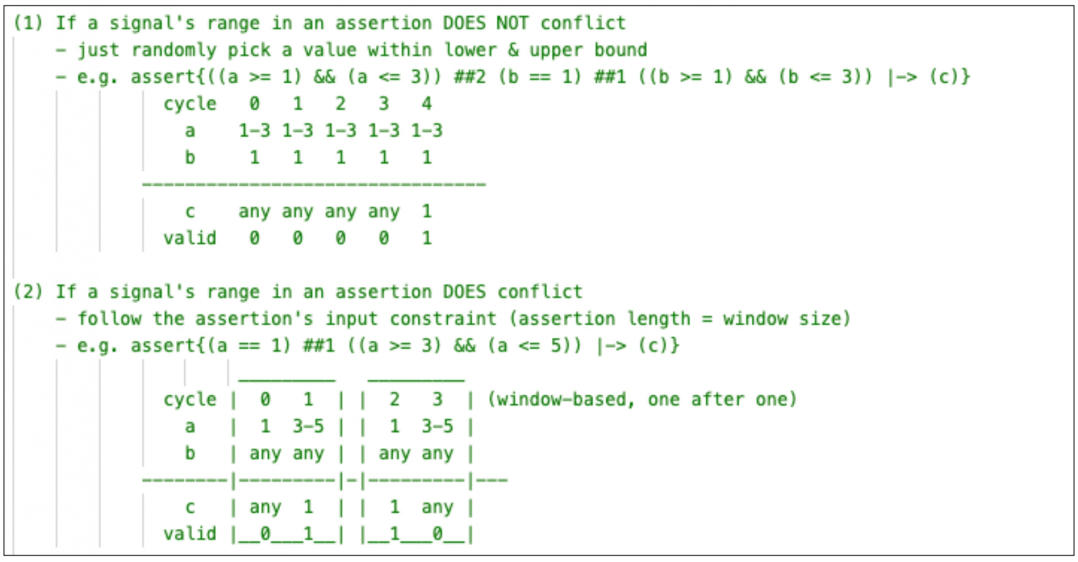



Figure 6.1: The illustration of the concept behind the "Generate Assertion Pattern Algorithm"

After generating the input patterns, we need to feed them into both the reference model and the DUV. In Alg. 4, we address the considerations when inputting patterns into


doi:10.6342/NTU202403664

the reference model. Initially, to filter assertions, we use Alg. 3 to generate customized input patterns and apply them to the reference model (lines 5-6 in Alg. 2). For the second filtering phase, we perform purely random simulations (line 13 in Alg. 2) instead of generating patterns based on each assertion's condition. Thus, we set a boolean value "$Random = True$" in Alg. 4 to directly generate and apply random patterns to the reference model.

In Alg. 4, if "$Random = True$" is specified, we initialize the random pattern length. For DUVs with one-cycle periods without input valid signals, we set the pattern length to match the designers' provided *golden_fail.csv* (lines 3-4). For DUVs with multiple-cycle periods, we set the random pattern length to the maximum transition from 0 to 1 (i.e., activation) among all input valid signals (lines 5-6).

Each input signal already has corresponding patterns, requiring slight modifications before applying them to the reference model. If there are output valid signals in the DUV, we generalize by feeding the input pattern to the DUV at every cycle, including during calculation and output result processes. In lines 19-24, we insert an input buffer into the DUV's pattern by duplicating the input at the end of the pattern to ensure the DUV does not prematurely read the next input set during the calculation stage (Fig.6.2). This allows us to apply the original pattern to the reference model and the buffer-inserted pattern to the DUV, ensuring that the read-in input can be matched.

Finally, the input patterns are first processed through an adapter to convert from the designers' format to our tool's defined format (line 25). The specifics of this format are detailed in the Appendix. Using the script for the reference model specified by the "*-ref_model*" parameter in Section 4.1, we execute the command *./<ref_model> <in-*


doi:10.6342/NTU202403664

*put_pattern*> <*output_result*> to obtain the golden result for the input pattern (line 26). The output result is then passed through the adapter again to convert it back to our tool's format (line 27).

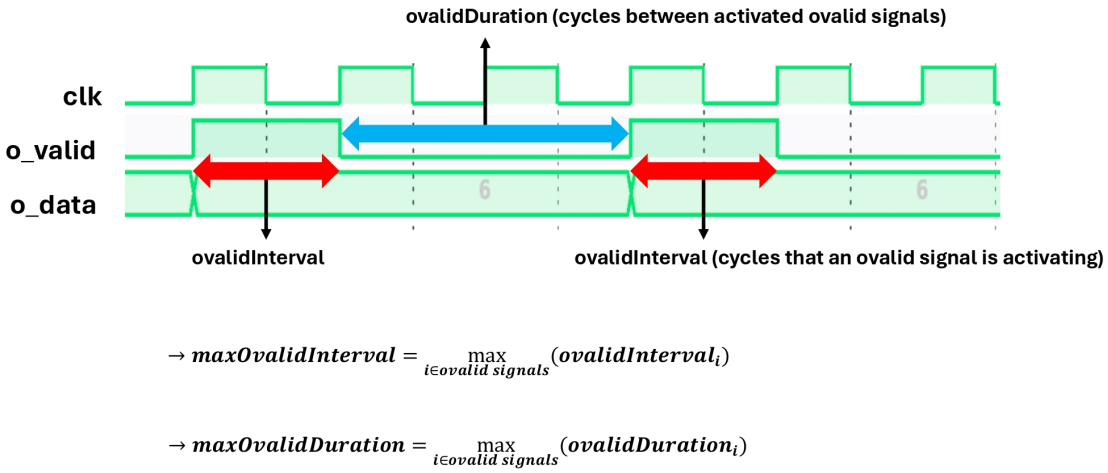



Figure 6.2: The illustration of the concept of "output valid interval" and "output valid duration"

## 6.3 Automatic Testbench Generation Algorithm

In addition to applying the input pattern to the reference model, we also need to apply it to the DUV and compare the results between $refResult$ and $duvResult$. We do not require designers to provide a testbench, as this could be inconvenient and may not align with our generalization standards. Therefore, we propose Alg. 5 to automatically generate a testbench based on the designers' settings in the configuration file. The categories for RTL designs in Section 3.5 and the configuration settings in Section 4.1 support this algorithm, ensuring it is generalized and capable of generating complicated blocks, such as input and output valid signal control conditions.


doi:10.6342/NTU202403664

**Algorithm 4** Hit Pattern to the Reference Model

**Input:** $inputPattern$, $Random$
**Output:** $refResult$

$ivalidNum \leftarrow$ # (input valid signals)
**if** $Random$ **then**
    **if** # (input valid signals) $== 0$ **then**
        $patternLength \leftarrow csvLength$
    **else**
        $patternLength \leftarrow$ *Get_Max_0-1_Transition_Num_of_All_IvalidSigs()*
**else**
    $patternLength \leftarrow$ ($patternLength$ from Alg. 3)
**for each** $sig_{ivalid}$ in $ivalidSigs$ **do**
    **if** # (input valid signals) $> 1$ **then**
        New an input pattern file for $sig_{ivalid}$
    **for** $j \leftarrow 1$ to $patternLength$ **do**
        **for each** $sig_{in}$ in $inputSigs$ **do**
            **if** ($sig_{in}$ is ivalid signal) **or** ($sig_{in}$ is don't care input signal) **or** ($sig_{in}$ is not controlled by $sig_{ivalid}$) **then**
                **continue**
            **if** $Random$ **then**
                $randPat \leftarrow$ *Generate_Pure_Random_Pattern(*$sig_{in}.width$*)*
                $inputPattern[sig_{in}]$.*push_back*($randPat$)
        **if** # (output valid signals) $> 0$ **then**
            $dupTime \leftarrow (2 \times maxOvalidInterval + maxOvalidDuration)$
            **if** (# (input valid signals) $\leq 1$) **and** (no given input activated constraint) **and** (no given testbench) **then**
                **if** $(j + 1)$ % $readInCycle == 0$ **then**
                    **for** $k \leftarrow 1$ to $dupTime$ **do**
                        Append the last input pattern to the end of $inputPattern[sig_{in}]$
Convert $inputPattern$ format by the adapter (*flowIn2userIn*)
Hit $inputPattern$ to the reference model
Convert $refResult$ format by the adapter (*userOut2flowOut*)
**return** $refResult$


doi:10.6342/NTU202403664

**Algorithm 5** Generate Testbench and Hit Pattern

**Input:** $inputPattern$
**Output:** $duvResult$

**if** has given testbench **then**
    Execute the given testbench under $inputPattern$
    **return** $duvResult$
**else**
    $dupNum \leftarrow (2 \times maxOvalidInterval + maxOvalidDuration - 1)$
    **if** $readInCycle > 1$ **then**
        $dupNum \mathrel{+}= (readInCycle - 1)$
    *Ofs_Timescale_and_Define()*
    *Ofs_and_Patch_Included_File()*
    *Ofs_Module_Sig_Memory_Declaration()*
    *Ofs_and_Initialize_Counter_and_Flag()*
    *Ofs_Vcd_Dumping_and_Indata_Memory()*
    *Ofs_Instantiation_of_the_Top_Module()*
    *Ofs_Clock_Reset_Generation()*
    **if** (has given input activated constraint) **or** (# (input valid signals) $> 1$) **then**
        **for each** $sig_{ivalid}$ **without** constraint **do**
            Activate $sig_{ivalid}$ first when the reset signal is not active
            Deactivate $sig_{ivalid}$ when reaching $readInCycle$
            Activate $sig_{ivalid}$ when $countOvalidDone ==$ # (output valid signals)
        **for each** $sig_{ivalid}$ **with** constraint **do**
            *Set_Counter_for_Constraint($sig_{ivalid}$)*
        *Ofs_Input_with_Ivalid_Controlling_Block_Setting()*
        *Ofs_Output_with_Ovalid_Controlling_Block_Setting()*
    **else**
        *Ofs_Load_Data_Task()*
        *Ofs_Wait_Ovalid_Task()*
    Execute the generated testbench under $inputPattern$
    **return** $duvResult$


doi:10.6342/NTU202403664

In Alg. 5, we first check whether designers have provided a testbench in the configuration file. If a testbench is available, we execute it using our standardized command format with the $inputPattern$ generated in Alg. 3 to obtain $duvResult$ (lines 1-3). If no testbench is provided, we automatically generate one.

Initially, we set the $dupNum$ for the input buffer as explained in Alg. 4 (lines 5-7), and then write the basic information to the testbench file (lines 8-14). This includes timescale, variable definitions, testbench module, register and wire signals, counters and flags based on the DUV's category from Section 3.5, VCD file dumping, memory for storing input data, instantiation of the DUV's top module, and clock/reset generation.

Subsequent blocks, such as "load input" and "wait for output", depend on the DUV's category from Section 3.5. For multiple interacting input valid signals or input activation constraints determining dependency or limitations between signals (as shown in Fig. 4.1 (d)), we divide the input valid signals into two groups: those without constraints and those with constraints.

For the former group, which has no dependencies, we ignore the ordering of these signals and use a flag and counter to control their activation (lines 16-19). For the latter group, the dependency between input valid signals dictates the ordering of $if condition$ declarations for the counter and flag in the testbench's control block. We extract signal dependencies as shown in Fig. 6.3 and initiate the for-loop from the leaf node to the root node (lines 20-21).

Finally, after appending the control block for input valid and output valid signals (lines 22-23 and line 26) and the task for loading input patterns (line 25), we execute the generated testbench with the $inputPattern$ to obtain the simulation result $duvResult$.


doi:10.6342/NTU202403664

```
->  e.g. invalid_2 |-> invalid_3       - (a)     =>     (dependency) invalid_1 -> invalid_2 -> invalid_3
    e.g. invalid_1 |-> ##1 invalid_3   - (b)                                   -> invalid_3
    e.g. invalid_1 |-> ##1 invalid_2   - (c)
    e.g. other_sig  |-> invalid_1      - (d)
-> invalid_2 = invalid_1 = invalid_3 = 0
-> by (a): invalid_2 = 0 ; invalid_3 = 1
   by (b): invalid_1 = 0 ; invalid_3 = 2
   by (c): invalid_2 = 1 ; invalid_3 = 3 (due to "invalid_2->next is invalid_3")
   by (d): invalid_1 = 0 (because antecedent is not invalid signal)
```

Figure 6.3: The illustration of the signal dependency among input activated constraints

The algorithm might seem abstract, so we provide examples to illustrate the generated testbench results.

Fig. 6.4 shows the basic declaration of the testbench, referenced in lines 8 to 14 in Alg. 5. In Fig. 6.5, lines 25 to 26 of Alg. 5 are demonstrated for a DUV without input activation constraints. Before reaching the specified pattern number, the while-loop continuously loads input data and waits for the calculation until the output valid signal is activated. In the $do\_load\_data$ task, input data is read by extracting the pattern based on its order in the DUV declaration (detailed format in the Appendix) and signal width. The $countDup$ variable is used to skip the remaining input buffer if the DUV finishes outputting early. In the $do\_wait\_ovalid$ task, the while-loop executes until all output valid signals are activated, ensuring all results are outputted. This prepares the system to skip the rest of the input buffer and get ready for the next input pattern set.

In Fig. 6.6 (lines 15-23 in Alg. 5), we first parse the input activation constraints to determine the required cycles for a signal to go high after another signal is high. This


doi:10.6342/NTU202403664

information allows us to extract the dependency of the input valid signals, as shown in Fig. 6.3, and set counters to activate signals after the specified number of cycles. Once the input valid signals are set to control the input data, we define tasks for loading input data and waiting for output valid signals until all input patterns are fed in.

After generating the testbench, it can be executed to obtain the simulation result in a VCD file, which will be used in Section 6.4 to compare with the reference model's golden result.

```
`timescale 1ns/100ps
`define CYCLE  10.0 // CLK period
`define HCYCLE (`CYCLE/2)
`define RST_DELAY 2

`define INFILE "input/alu_cvsd-hw1_bug_inst-101/temp/duv_in.dat"
`define DUV_OUTFILE "input/alu_cvsd-hw1_bug_inst-101/temp/alu.vcd"

module testbed;

reg i_clk, i_rst_n;
reg i_valid;
reg [7:0] i_data_a;
reg [7:0] i_data_b;
reg [2:0] i_inst;
wire o_valid;
wire [7:0] o_data;
reg [18:0] indata_mem [0:4000];

integer cycle;
integer countDup;
integer PAT_NUM, pat_num_idx;
integer indata_idx;

integer count_done;
integer flag_o_valid;

initial begin
    $dumpfile(`DUV_OUTFILE);
    $dumpvars();
end
```

```
alu u_alu (
    .i_clk(i_clk),
    .i_rst_n(i_rst_n),
    .i_valid(i_valid),
    .i_data_a(i_data_a),
    .i_data_b(i_data_b),
    .i_inst(i_inst),
    .o_valid(o_valid),
    .o_data(o_data)
);

initial $readmemb(`INFILE, indata_mem);

initial begin
    PAT_NUM = 0;
    while (indata_mem[PAT_NUM] !== 19'bx)
        PAT_NUM = PAT_NUM + 1;
    $display("PAT_NUM = %d", PAT_NUM);
end

initial begin
    i_clk = 0;
    i_valid = 0;
    i_data_a = 0;
    i_data_b = 0;
    i_inst = 0;
    i_rst_n = 0; # (`RST_DELAY * `CYCLE);
    i_rst_n = 1;
end
always begin #(`CYCLE/2) i_clk = ~i_clk; end
```

Figure 6.4: The example of generated testbench (basic declaration)

## 6.4 Results Comparison with the Reference Model

So far, we have obtained the $inputPattern$, $duvResult$, and $refResult$. In Alg. 6, our goal is to determine whether the given $assertion$ is suspected of containing a bug by comparing the results between the DUV and the reference model. We will classify the assertion as $golden$, $failed$, or $invalid$ to indicate its status for the final bug localization phase. If the state is $golden$, the assertion holds true for both the DUV and the reference


doi:10.6342/NTU202403664

```
initial begin
    cycle = 0;
    indata_idx = 0;
    pat_num_idx = 0;
    countDup = 0;
    count_done = 0;
    flag_o_valid = 0;
    while (pat_num_idx < PAT_NUM) begin
        // load input
        do_load_data;
        // wait ovalid
        i_valid = 0;
        do_wait_ovalid;
        countDup = 0;
        // next
        cycle = cycle + 1;
        pat_num_idx = pat_num_idx + 1;
    end
    $finish;
end
```

```
task do_load_data; begin
    i_valid = 1;
    i_data_a = (indata_mem[indata_idx] >> 0) % 256;
    i_data_b = (indata_mem[indata_idx] >> 8) % 256;
    i_inst = (indata_mem[indata_idx] >> 16) % 8;
    countDup = countDup + 1;
    indata_idx = indata_idx + 1;
    @(posedge i_clk);
end endtask

task do_wait_ovalid; begin
    while ((count_done < 1) && (countDup !== 4)) begin
        if ((o_valid == 1) && (flag_o_valid == 0)) begin
            flag_o_valid = 1;
            count_done = count_done + 1;
        end
        if ((flag_o_valid == 1) && (o_valid == 0)) begin
            flag_o_valid = 0;
        end
        cycle = cycle + 1;
        countDup = countDup + 1;
        pat_num_idx = pat_num_idx + 1;
        indata_idx = indata_idx + 1;
        @(posedge i_clk);
    end
    // skip all duplicate buffer pattern
    while ((count_done == 1) && (countDup !== 4)) begin
        countDup = countDup + 1;
        pat_num_idx = pat_num_idx + 1;
        indata_idx = indata_idx + 1;
    end
end endtask
endmodule
```

Figure 6.5: The example of generated testbench (load data, wait ovalid task)

```
always @(posedge i_clk) begin
    cycle = cycle + 1;
    // activate ivalid signal
    if (count_for_i_op_valid == 1) begin
        count_for_i_op_valid = 0;
        i_op_valid = 1;
    end
    if (count_for_i_op_valid == 0) begin
        if ((o_op_ready)) begin
            count_for_i_op_valid = count_for_i_op_valid + 1;
        end
        else begin
            count_for_i_op_valid = 0;
        end
    end
    if (count_for_i_in_valid == 1) begin
        count_for_i_in_valid = 0;
        i_in_valid = 1;
    end
    if (count_for_i_in_valid == 0) begin
        if (((i_op_valid) && (((indata_mem_1[indata_idx_1] >> 0) % 16) == 0))) begin
            count_for_i_in_valid = count_for_i_in_valid + 1;
        end
        else begin
            count_for_i_in_valid = 0;
        end
    end
```

```
    // start loading input
    if (flag_i_op_valid == 1) begin
        if (count_read_1 == 1) begin
            i_op_valid = 0;
            count_read_1 = 0;
            flag_i_op_valid = 0;
        end
    end
    if (i_op_valid == 1) begin
        i_op_mode = (indata_mem_1[indata_idx_1] >> 0) % 16;
        flag_i_op_valid = 1;
        count_read_1 = count_read_1 + 1;
        indata_idx_1 = indata_idx_1 + 1;
        pat_num_idx_1 = pat_num_idx_1 + 1;
    end
    if (flag_i_in_valid == 1) begin
        if (count_read_2 == 2048) begin
            i_in_valid = 0;
            count_read_2 = 0;
            flag_i_in_valid = 0;
        end
    end
    if (i_in_valid == 1) begin
        i_in_data = (indata_mem_2[indata_idx_2] >> 0) % 256;
        flag_i_in_valid = 1;
        count_read_2 = count_read_2 + 1;
        indata_idx_2 = indata_idx_2 + 1;
        pat_num_idx_2 = pat_num_idx_2 + 1;
    end
    // finish
    if ((pat_num_idx_1 == PAT_NUM_1) && (pat_num_idx_2 == PAT_NUM_2)) begin
        $finish;
    end
end

endmodule
```

Figure 6.6: The example of generated testbench (has input activation constraint)



doi:10.6342/NTU202403664

model. If the state is $failed$, the assertion is suspected to contain a bug and will be our candidate for the final phase since it is true for the DUV but false for the reference model (spec). Assertions classified as $invalid$ will not be considered, as their status regarding the bug cannot be determined.

In Alg. 6, it's important to understand some key concepts. The algorithm addresses two scenarios: DUVs without output valid signals and DUVs with output valid signals.

In the first scenario (DUVs without output valid signals, lines 1-31 of Alg. 6), we typically deal with a one-cycle period DUV since there's no output valid signal to control the output result. Consequently, the output data in every cycle is relevant. Ideally, if the DUV is bug-free, its results should match the reference model's results at every cycle.

We begin with the given assertion (e.g. "$req1 \mid\!\rightarrow gnt1$" as shown in the top-right of Fig. 6.7). Since this assertion is verified as true for the DUV by VC Formal, we need to compare whether its implication holds true for the reference model. Note that we disregard internal signals of the DUV in the assertion when comparing waveforms with the reference model because the reference model is treated as a black box. It only concerns itself with the correctness of the output results, not the internal operations of the DUV.

Moreover, we do not assume any naming conventions between the DUV and the reference model's output signals; instead, the output signals will be matched by the SDM algorithm discussed in Chapter 4. In this scenario, the DUV and the reference model share the same timeline. When an input pattern set is fed in, both the DUV and the reference model will output the result in the same cycle. We can then verify each assertion's implication using the input signal values (as shown on the left side of Fig. 6.7) and output signal values (the right-middle of Fig. 6.7 for the DUV's waveform and the right-bottom


doi:10.6342/NTU202403664

**Algorithm 6** Compare with the Reference Model

**Input:** $assertion$, $inputPattern$, $duvResult$, $refResult$
**Output:** $state$ ($golden = 1, failed = 2, invalid = 3$)

**if** # (output valid signals) == 0 **then**
  $assertionDepth \leftarrow$ *Parse_Assertion_Cycle*($assertion$)
  $antExpected \leftarrow$ *Store_Antecedent_Expected_Range*($assertion$)
  $consExpected \leftarrow$ *Store_Consequent_Expected_Range*($assertion$)
  $found \leftarrow False$
  $golden \leftarrow False$
  **for** $i \leftarrow 1$ to $patternLength$ **do**
    $satisfy \leftarrow True$
    **for** $j \leftarrow 1$ to $assertionDepth$ **do**
      **for each** $ant$ in $antExpected$ **do**
        $lowerBound \leftarrow ant$.second[$j$].first
        $upperBound \leftarrow ant$.second[$j$].second
        $goldenVal \leftarrow name2BinResult$[$ant$.first][$i + j$]
        **if** *Out_of_Range*($goldenVal, lowerBound, upperBound$) **then**
          $satisfy \leftarrow False$
          **break**
      **if** $satisfy == False$ **then break**
      **if** $j == (assertionDepth - 1)$ **then**
        $found \leftarrow True$
        **for each** $cons$ in $consExpected$ **do**
          $lowerBound \leftarrow cons$.second[$j$].first
          $upperBound \leftarrow cons$.second[$j$].second
          $goldenVal \leftarrow name2BinResult$[$cons$.first][$i + j$]
          **if** *Out_of_Range*($goldenVal, lowerBound, upperBound$) **then**
            $golden \leftarrow False$
            **break**
    **if** $golden == False$ **then break**
  **if** $found == False$ **then return** $invalid$
  **else**
    **if** $golden == True$ **then return** $golden$
    **else return** $failed$
**else**
  *Extract_Waveform_Timeline*($isDUVFailVCD = True$)
  *Extract_All_Ovalid_Output*($isDUVFailVCD = True$)
  $state \leftarrow$ *Heuristic_Method_for_Assertion_Filtering*($assertion$)
  **if** $state == failed$ **then return** $failed$
  **if** *Check_Assertion_on_Simulation_Trace*($duvResult$) **then**
    *Extract_Waveform_Timeline*($isDUVFailVCD = False$)
    *Extract_All_Ovalid_Output*($isDUVFailVCD = False$)
    $state \leftarrow$ *Heuristic_Method_for_Assertion_Filtering*($assertion$)
  **return** $state$


doi:10.6342/NTU202403664

for the reference model's waveform) as needed.

For example, in the 16-th pattern, when $req1 = 1$, the DUV's $gnt1$ is 1 as implicated by the assertion, but the reference model's $gnt1$ is 0, violating the implication. Therefore, we mark this assertion as suspected or in a "*failed*" state.

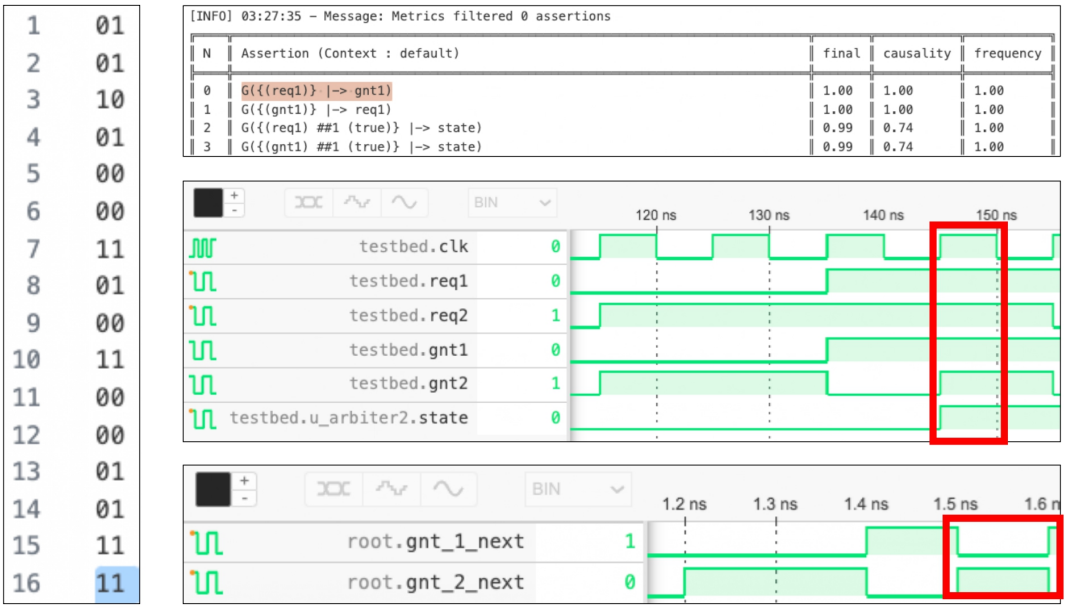



Figure 6.7: The example of comparing the results between the reference model and the DUV (no output valid signal)

In the second scenario (DUVs with output valid signals, lines 32-41 of Alg. 6), the reference model outputs results in the same cycle it reads the input data. However, the DUV spans multiple cycles for calculations before outputting the result, making cycle-by-cycle comparison impractical. Therefore, we use the concept of UVM's monitor and scoreboard (Fig. 6.8). The monitor detects the DUV's output result when the output valid signal is activated, and the scoreboard pops the next output result from the reference model to compare with the DUV's result. This approach addresses the timeline mismatch between the DUV and the reference model.

Fig. 6.9 illustrates this concept. The top part of the figure shows the DUV's waveform, and the bottom part shows the reference model's waveform. In this example, the


doi:10.6342/NTU202403664

output valid signal of the DUV is $serviceTypeOut$. When $serviceTypeOut == 0$, the output result is valid (indicating the FSM state is $off$ in the DUV). When the first output valid signal of the DUV is activated (highlighted in red in the top figure), the DUV's output signal values (e.g. $itemTypeOut, coinOutNTD\_1, coinOutNTD\_5$) at that cycle are compared with the first output result of the reference model (red block in the bottom figure). Similarly, when the second output valid signal of the DUV is activated (highlighted in purple in the top figure), the DUV's output signal values at that cycle are compared with the second output result of the reference model (purple block in the bottom figure). This method is called "comparison at the transaction level".

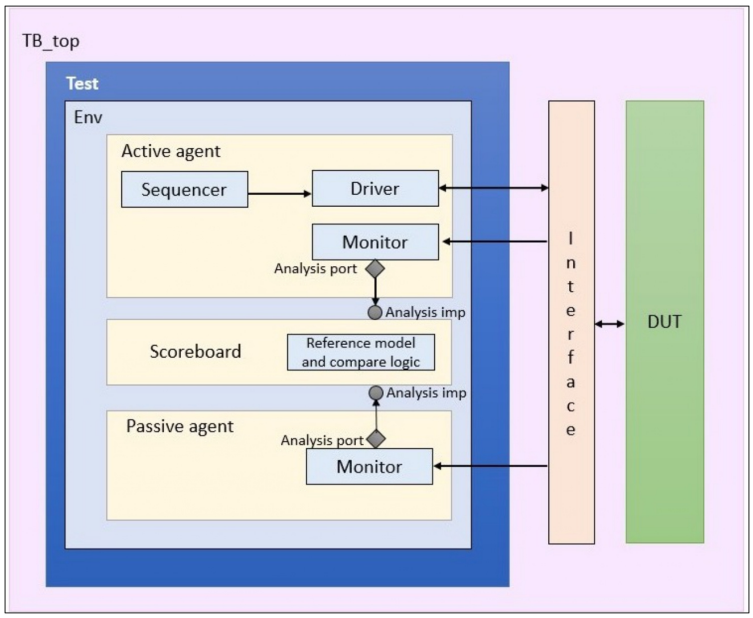



Figure 6.8: The concept of comparing the results between the reference model and the DUV using UVM's monitor and scoreboard (has output valid signals)

After comparing at the transaction level, we determine the state ($golden$, $failed$, or $invalid$) of the assertion (lines 35 and 40 of Alg. 6). The concept behind this state categorization method is illustrated in Fig. 6.10.

At the top of the figure, we first extract the waveform that satisfies the given assertion in the DUV's simulation result ($duv\_fail.vcd$ in line 33 of Alg. 6; $duvResult$ in line 38


doi:10.6342/NTU202403664


Figure 6.9: The example of comparing the results between the reference model and the DUV (has output valid signals)

of Alg. 6). We assume the range marked "$assertion$" in Fig. 6.10 satisfies the assertion's implication. Next, we identify the previous output valid signal that was activated right before the beginning of the assertion's waveform ($prev$) and the next output valid signal activated right after the end of the assertion's waveform ($next$). These points are used to determine the assertion's state (lines 34-35 and lines 39-40 of Alg. 6). Note that $prev$ and $next$ activated output valid signals exclude the assertion's waveform.

The bottom of Fig. 6.10 explains the rationale behind choosing these two points. In RTL design, when a waveform satisfies an assertion's implication, it indicates that a specific condition or event has occurred during simulation or verification. This means that the behavior related to that assertion meets the specified criteria, either confirming expected behavior or highlighting unexpected situations that need further inspection.

Using the transaction comparison technique mentioned above, we compare the results between the DUV and the reference model at $prev$ and $next$. The assertion's state is determined as follows:


doi:10.6342/NTU202403664

- If the comparison result pair for $prev$ and $next$ is $(True, True)$, the assertion is marked as "$golden$" since no unexpected behavior occurs.
- If the comparison result is $(True, False)$ or $(False, True)$, the assertion is marked as "$failed$" (suspected) because unexpected behavior is detected when passing through the assertion's waveform, warranting further analysis.
- If the comparison result is $(False, False)$, the assertion is marked as "$invalid$" since the DUV's behavior is consistently unexpected, making it impossible to determine its relevance to the assertion.

If there is no $prev$ (i.e. from the beginning of the DUV's waveform to the assertion's waveform, no output valid signal is activated) or no $next$ (i.e. from the end of the assertion's waveform to the end of the DUV's waveform, no output valid signal is activated), we consider the comparison result for $prev$ and $next$ as "$True$".

In Alg. 6, we first extract the waveform from the provided $duv\_fail.vcd$. Since the assertion is derived from this simulation result file, we can find at least one segment of the waveform that satisfies the assertion's implication (lines 33-36). Next, we use HARM's feature to check if the assertion's waveform appears in our constrained random simulation result ($duvResult$). If it does, we proceed to further analyze assertions that cannot be conclusively identified as suspected (lines 37-40).

After determining the state of each assertion in Alg. 6, we retain the assertions in the "$failed$" state as suspected bug candidates and proceed to the final phase of our process: Bug Localization in RTL Codes.


doi:10.6342/NTU202403664

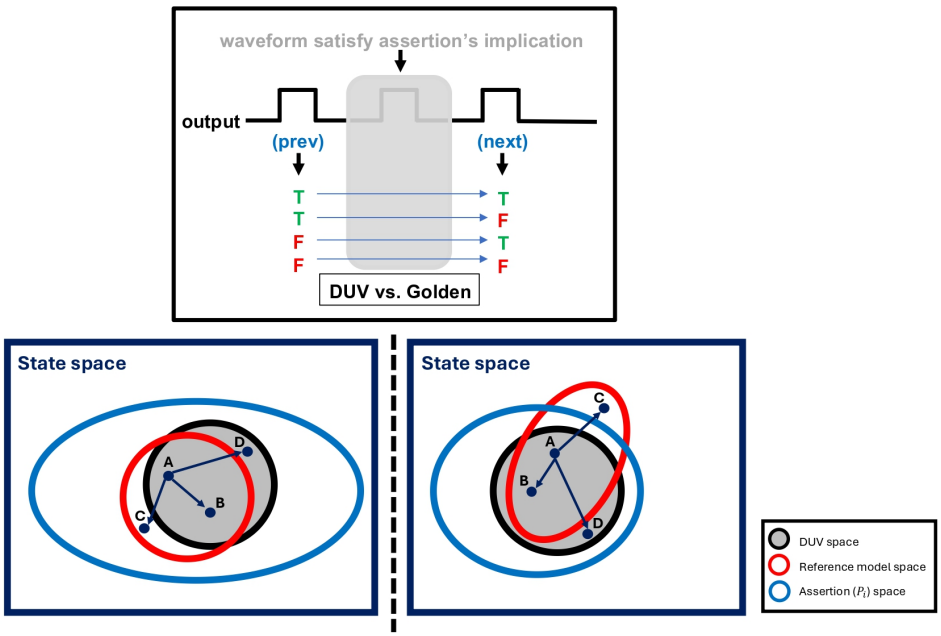



Figure 6.10: The illustration of the concept behind the "Heuristic Method for Assertion Filtering Algorithm"



doi:10.6342/NTU202403664

# Chapter 7 Bug Localization Phase : Bug Localizing in RTL Codes

In this chapter, we begin with an exploration of the Bug Localizing Algorithm in Section 7.1, where we discuss how our tool maps suspected assertions to specific RTL code lines. This mapping is crucial for pinpointing the exact location of potential bugs.

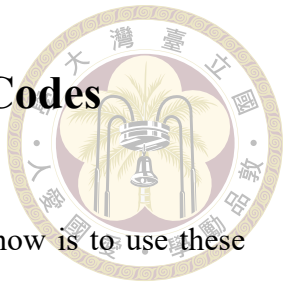
In Section 7.2, we introduce the Hint Signal Weight Formula, which prioritizes signals based on their likelihood of being related to the bug. By assigning weights to these signals, we can guide designers to the most promising areas of the design more quickly and efficiently.

Finally, in Section 7.3, we present the bug Localization results and debug hints for designers. This section consolidates the findings from the previous sections, offering clear and concise debug hints. These hints are intended to help designers focus their efforts on the most suspected control paths, thus significantly reducing the time and effort required to identify and fix bugs.

Through this chapter, we aim to equip designers with the tools and methodologies necessary to tackle bug localization with greater precision and confidence.


doi:10.6342/NTU202403664

## 7.1 Mapping Suspected Assertions to RTL Codes



Having generated suspected assertions in Chapter 6, our goal now is to use these suspect assertions to map back to the DUV's code and localize the potential bug positions for designers. We will record the hint information in the output file $Final\_Bug\_Zone.txt$.

**Algorithm 7** Bug Localizing Algorithm

**Input:** $suspectedAssertions$
**Output:** $Final\_Bug\_Zone.txt$

```
duvStructuralInfo ← Extract_AST()
for each 𝔸 in suspectedAssertions do
    for each ℂ in allControlPaths do
        if Check_Antecedent_in_Control_Path(𝔸, ℂ) then
            hintSignals ← hintSignals ∪ (the matched antecedent signals in 𝔸)
            suspectCandSets ← suspectCandSets ∪ ℂ
hintSignals ← Sort_Hint_Signal_Weight(hintSignals)
suspectBins ← Categorize_Suspected_Bins(suspectCandSets)
```

Before proceeding, our algorithm for mapping suspected assertions back to the RTL lines is inspired by [1]. In that paper, the authors also use their algorithms to filter suspected assertions, and in their final step, they employ static analysis to map these suspected assertions back to the design. For each suspected assertion, their goal is to check whether any antecedent signals in the assertion are in the control path, where "in the control path" is defined as : (1) The sensitivity list or right-hand side of the assignments in the **always** block, (2) The conditional statement of **if** block, (3) The case condition statement of **case** block.

If the antecedent signal of an assertion is located within a control path block, that block will be flagged as a final suspected control path potentially containing the bug. To facilitate efficient debugging, the authors have categorized these suspected control path blocks into four bins, prioritized in descending order of likelihood that they contain the


doi:10.6342/NTU202403664

bug. These categories are intended to guide designers to the most probable bug locations first. The four categories of suspected bins are listed below, and the categorization method is illustrated in Fig. 7.1.

1. $SuspBin_1$**: Matched *if* blocks within matched *always* block in the suspected control path**: These blocks are given the highest priority because the nested $if$ blocks within $always$ blocks can significantly narrow down the bug location.

2. $SuspBin_2$**: Remaining matched *if* blocks in the suspected control path**: These blocks have a lower priority than $SuspBin_1$, as there is no multi-level hierarchy of control paths all pointing to the potential bug location.

3. $SuspBin_3$**: Remaining matched *case* blocks in the suspected control path**: $case$ statements typically cover broader ranges in RTL codes than $if$ statements, making the identified bug zone within a $case$ statement less precise.

4. $SuspBin_4$**: Remaining matched *always* blocks in the suspected control path**: $always$ blocks cover the widest range in RTL codes, thus having the lowest priority for bug localization efforts.

Let's revisit Alg. 7. First, we will use Pyverilog to extract the structural information of the DUV (line 1), including the begin and end line numbers of each control path block (e.g. always, if/case statements), the signal names in the control path's condition statements, and so on. This structural information, referred to as the abstract syntax tree (AST), forms the foundation for our analysis. Next, for each suspected assertion, we will check whether the antecedent signal of the suspected assertion is present in any control path of the DUV. If it is, we will store the antecedent signal in a set named $hintSignals$ and the corresponding


doi:10.6342/NTU202403664

```
module dummy
    always_1: begin
        if_1: begin
        end
        case_1: begin
        end
    end
    always_2: begin
        if_2: begin
        end
        case_2: begin
        end
        if_3: begin
        end
        case_3: begin
        end
    end
    always_3: begin
    end
endmodule
```

| $SuspCand$ | $SuspBin_1$ | $SuspBin_2$ | $SuspBin_3$ | $SuspBin_4$ |
|---|---|---|---|---|
| $always\_1$<br>$always\_3$<br>$if\_1$<br>$if\_2$<br>$case\_1$<br>$case\_2$ | $if\_1$ | $if\_2$ | $case\_1$<br>$case\_2$ | $always\_3$ |

Figure 7.1: The example illustrating the categorization method described in [1]

suspected control path in a set named $suspectCandSets$ (lines 2-6). After traversing all suspected assertions, we will sort the hint signals based on our metrics (discussed in Section 7.2) to provide guidance for designers (line 7). Additionally, we will categorize the suspected control paths into four bins using the method explained above, which will also be discussed in Section 7.3 (line 8). Finally, we will generate the $Final\_Bug\_Zone.txt$ file based on $hintSignals$ and $suspectBins$ to guide designers in their debugging process.

## 7.2 Weighting Hint Signals

In this section, we will introduce our metrics for ranking the suspected degree of hint signals. For each hint signal, we will evaluate its importance in relation to the bug from three aspects: **Assertion candidate count** ($factor_A$), **Parent dependency** ($factor_P$), and **Distribution** ($factor_D$).


doi:10.6342/NTU202403664

$$factor_A = \frac{\sqrt{\#\ suspected\ assertions\ that\ contain\ Sig_i}}{\sum_{Sig_i \in hint\ signals} \sqrt{\#\ suspected\ assertions\ that\ contain\ Sig_i}} \quad (7.1)$$

For $factor_A$, we consider the magnification of the suspected degree of each signal after two-level filtering for assertions (Chapter 5 and Chapter 6). If a hint signal is more related to the bug, its unexpected behavior in the assertions containing it will have a higher probability of passing our suspected assertion filtering algorithm in Chapter 6 than other assertions. Thus, as shown in Equation (7.1), the general idea for this factor is to calculate the ratio of the number of suspected assertions containing the hint signal to the total number of suspected assertions. This represents the magnification of the suspected degree of each signal.

Our formula incorporates some modifications for accuracy. In the denominator, we sum up all the numbers of suspected assertions containing hint signals rather than directly using the number of suspected assertions. This approach takes validity into account since there may be many assertions with antecedent signals that are not hint signals of interest. Including these assertions could dilute the score of $factor_A$ with meaningless data.

For both the denominator and numerator, we apply the square root to the number of suspected assertions. This adjustment is necessary because our focus is on the hint signals rather than the assertions themselves. An assertion is composed of propositions, each involving a numerical operation on a hint signal (e.g. $a \geq 1$, $b == 0$). Since our assertion candidates are generated by HARM, and as discussed in HARM's paper [29], the worst-case scenario for mining assertions is exponential in relation to the temporal and proposition sizes. Therefore, the dimension of a hint signal in an assertion should


doi:10.6342/NTU202403664

theoretically be logarithmic with respect to the number of assertions. However, the paper also mentions that in practical scenarios, the worst case rarely occurs due to heuristic methods that select a small set of possible decisions at each level of the decision tree. Hence, instead of applying a logarithm, we opt for a square root to avoid overly pessimistic calculations.

$$factor_P = \sqrt[\frac{m}{4}]{\prod_{i=1}^{m} \frac{1}{\#\ hint\ signals\ in\ control\_branch_i}},\ m = \#\ control\_branch \quad (7.2)$$

For $factor_P$, we consider the hierarchy of control paths. Imagine a scenario where a hint signal $sig_1$ is in the sensitivity list of an $always$ block, within which there is a $case$ statement with several states and another hint signal $sig_2$ in its condition statement. Additionally, inside the $case$ statement, there is an $if$ condition with hint signal $sig_3$ in its condition statement. Intuitively, it is less precise to tell designers the bug might be in the control path containing $sig_1$ (always block) or $sig_2$ (case statement) compared to $sig_3$ (if block). Therefore, to reduce the importance of a hint signal that spans a broader range, we penalize it by calculating the reciprocal of the depth (number of other control paths) under the control path containing the target hint signal, which constitutes its $factor_P$ score.

Before proceeding, we need to introduce a term used in our tool, illustrated in Fig. 7.2. In Fig. 7.2, we use the line numbers of each control path block to form a dependency graph, where a child node represents a control path contained within its parent node control path. This nested always/if/case statement structure is termed a “control branch”, which is a path from the root node to the leaf node. The root node must be an $always$ block in the DUV, and all control branches extending from the root node form a “control tree”. A


doi:10.6342/NTU202403664

DUV might be composed of several $always$ blocks, so all control trees combined form a "control forest", constituting the DUV's architecture.

In the formula, we not only apply the reciprocal to the hint signal's score, but also consider that a hint signal might appear in several segments of control branches. Therefore, we calculate the geometric average of the scores for each segment of control branches for the hint signal, as the geometric average is used to find the average among ratios. We aim to differentiate between hint signals with or without hierarchy. Specifically, the score difference between ($\#\ hint\ signals\ in\ control\_branch_i = 1$) and ($\#\ hint\ signals\ in\ control\_branch_i = 2$) should be notable, but the gap between ($\#\ hint\ signals\ in\ control\_branch_i = 2$) and ($\#\ hint\ signals\ in\ control\_branch_i > 2$) should not be too large. Thus, we apply an additional fourth root to the $factor_P$ score, with $(\frac{1}{1})^{\frac{1}{4}} - (\frac{1}{2})^{\frac{1}{4}} \cong 2 \times ((\frac{1}{2})^{\frac{1}{4}} - (\frac{1}{3})^{\frac{1}{4}})$.

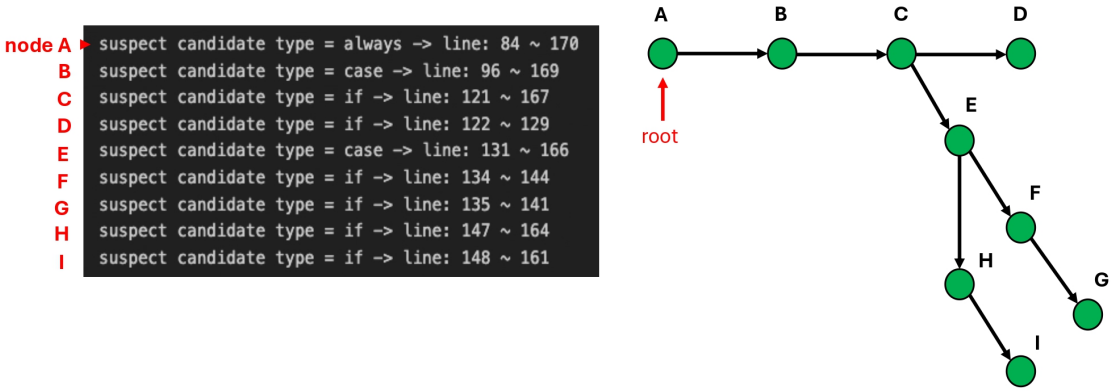
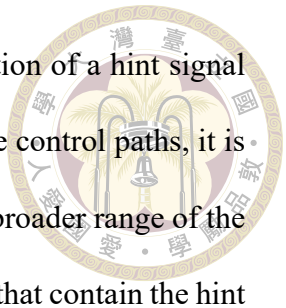



Figure 7.2: The illustration of the concept of control branches in $factor_P$

$$factor_D = \frac{\#\ control\ paths\ that\ contain\ Sig_i}{\#\ control\ paths\ in\ the\ DUV},\ Sig_i \in hint\ signals \qquad (7.3)$$



doi:10.6342/NTU202403664



Lastly, $factor_D$ is an intuitive metric. We consider the distribution of a hint signal within the control paths of the DUV. If a hint signal is present in more control paths, it is more likely to be associated with the bug, as it is distributed across a broader range of the DUV. Therefore, we directly calculate the percentage of control paths that contain the hint signal relative to the total number of control paths.

$$hint\ signal\ weight = \alpha \cdot factor_A + \beta \cdot factor_P + \gamma \cdot factor_D \tag{7.4}$$

Thus, we define the formula to calculate the suspected degree (hint signal weight) for each hint signal in Equation (7.4). This formula is a weighted sum of the three factors discussed earlier, with all factors normalized to a range from 0 to 1 to eliminate the effects of different units or scales. Consequently, the hint signal weight will also be a real number between 0 and 1. In the formula, the weights $\alpha$, $\beta$, and $\gamma$ correspond to $factor_A$, $factor_P$, and $factor_D$, respectively. Empirically, we set $\alpha = 0.45$, $\beta = 0.35$, and $\gamma = 0.2$, which aligns with our approach.

Firstly, $factor_A$ is given the highest weight because our tool emphasizes assertion-based verification, and the previous chapters' algorithms focus on filtering suspected assertions. Therefore, magnifying the impact of assertions is crucial, justifying the higher weight for $factor_A$. Secondly, $factor_P$ is also significant because hierarchical penalties are important for the accuracy of the final bug localization results. For example, in an $always$ block, if a $case$ statement contains many states, and within one state an $if$ statement determines operations, the range of the $case$ or $always$ block is too broad compared to the $if$ statement. Hence, hint signals in the $case$ or $always$ blocks should be penalized to reduce their importance. Lastly, $factor_D$ is assigned the least weight. While it is logi-


doi:10.6342/NTU202403664

cal that a hint signal distributed over a broader range in the DUV is more likely to contain a bug, this is not a critical indicator for debugging. It is a probabilistic issue, so we do not prioritize this factor heavily.

After calculating the hint signal weight for all hint signals, we will sort them in descending order to guide designers in debugging, which will be discussed in the next section.

## 7.3 Bug Localization Results and Debugging Hints for Designers

In the end of Alg. 7, our tool outputs a "*Final_Bug_Zone.txt*" file for designers, providing hints to guide them in debugging the most suspected parts. In Fig. 7.3, we initially provide a priority list of hint signals, ranked by their hint score calculated in Section 7.2, to direct designers in searching the related control paths. For example, designers can first investigate control paths containing the hint signal "$i\_data\_a$", which has the highest hint score in our algorithm. As shown in Fig. 7.4, designers can use the suspected control paths extracted by the algorithm in Section 7.1 to focus on those containing the hint signals from Fig. 7.3 in order. For instance, with "$i\_data\_a$" prioritized, they can search the extracted control paths with "$i\_data\_a$" in the condition statement, leading them to quickly locate the bug.

In Fig. 7.5, we provide optional crucial information for designers who wish to understand the score calculation for the three factors. For example, the information for "*Factor A*" lists the number of suspected assertions containing each hint signal in this phase. For "*Factor P*", we provide the hierarchical depth for each hint signal, as explained in Section


doi:10.6342/NTU202403664

```
1   ================================================================
2                    Bug Localization Suspect Zone
3   ================================================================
4
5   ** HINT Result **
6   -> The score is calculated by 3 factors
7   -> The higher the signal's hint score is, the more suspect the signal's control path is
8      (the control paths of the highest score signal can be checked first)
9   ------------------------------------------------------------------------------------------
10  | signal name           | hint score            | factor A   | factor P   | factor D   |
11  ------------------------------------------------------------------------------------------
12  | i_data_a              | 0.5623                | 0.2496     | 1.0000     | 0.5000     |
13  ------------------------------------------------------------------------------------------
14  | i_data_b              | 0.5139                | 0.1975     | 1.0000     | 0.3750     |
15  ------------------------------------------------------------------------------------------
16  | o_data_w              | 0.4659                | 0.1186     | 1.0000     | 0.3125     |
17  ------------------------------------------------------------------------------------------
18  | i_inst                | 0.3827                | 0.1131     | 0.7071     | 0.1875     |
19  ------------------------------------------------------------------------------------------
20  | i_valid               | 0.3115                | 0.0179     | 0.5774     | 0.1875     |
21  ------------------------------------------------------------------------------------------
22  | o_data                | 0.1410                | 0.2855     | 0.0000     | 0.0625     |
23  ------------------------------------------------------------------------------------------
24  | temp_multiply         | 0.0875                | 0.0000     | 0.0000     | 0.4375     |
25  ------------------------------------------------------------------------------------------
26  | mod_reg               | 0.0250                | 0.0000     | 0.0000     | 0.1250     |
27  ------------------------------------------------------------------------------------------
28  | o_valid               | 0.0205                | 0.0179     | 0.0000     | 0.0625     |
29  ------------------------------------------------------------------------------------------
```

Figure 7.3: The example of sorted hint signals provided to designers

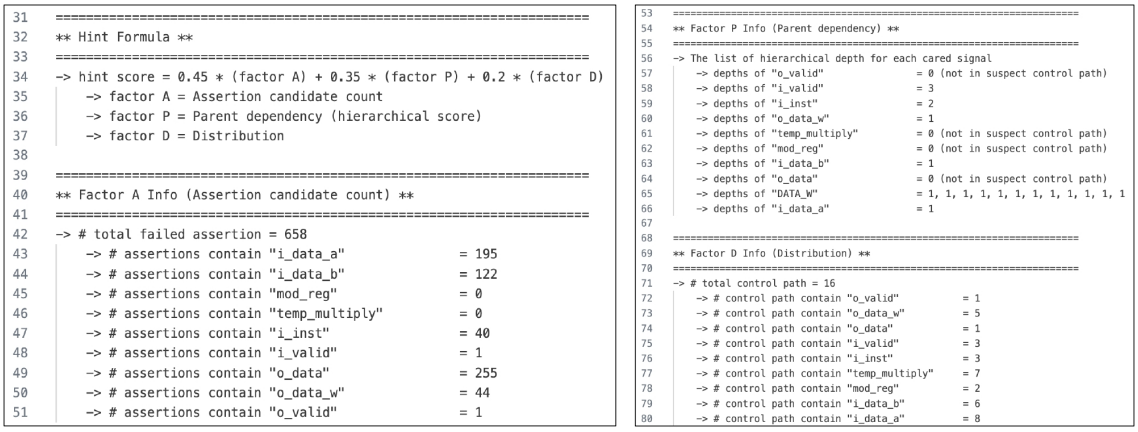
```
84  ------------------------------------------------------------------------
85  Suspect rank no.1 : matched "if" block(s) in matched always blocks
86  ------------------------------------------------------------------------
87  (1)
88  - LINE -> 51 ~ 53
89  - RTL CODES ->
90              else if(i_data_a[DATA_W-1] == 1'b1 && i_data_b[DATA_W-1] == 1'b1 && o_data_w[DATA_W-1] == 1'b0) begin
91                  o_data_w = 9'b010000000;
92              end
93  >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
94  (2)
95  - LINE -> 60 ~ 62
96  - RTL CODES ->
97              else if(i_data_a[DATA_W-1] == 1'b1 && i_data_b[DATA_W-1] == 1'b0 && o_data_w[DATA_W-1] == 1'b0) begin
98                  o_data_w = 9'b010000000;
99              end
100 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
101 (3)
102 - LINE -> 91 ~ 97
103 - RTL CODES ->
104             else if(i_data_a <= $signed(8'b11000000)) begin
105                 o_data_w = 9'b000000000;
106             end
107             else begin// 1/4x+1/2
108                 // o_data_w = {1'b0, (i_data_a >>> 2) + $signed(8'b00010000)};  // FIXME: golden -> arithmetic right shift (pad with sign bit)
109                 o_data_w = {1'b0, (i_data_a >> 2) + $signed(8'b00010000)};  // FIXME: bug -> logical right shift (pad with 0s)
110             end
111
```

Figure 7.4: The example of the suspected control path containing bugs



doi:10.6342/NTU202403664

7.2. Lastly, the information for "*Factor D*" shows how many control paths in the DUV contain each hint signal.



```
31  ======================================================================
32  ** Hint Formula **
33  ======================================================================
34  -> hint score = 0.45 * (factor A) + 0.35 * (factor P) + 0.2 * (factor D)
35      -> factor A = Assertion candidate count
36      -> factor P = Parent dependency (hierarchical score)
37      -> factor D = Distribution
38
39  ======================================================================
40  ** Factor A Info (Assertion candidate count) **
41  ======================================================================
42  -> # total failed assertion = 658
43      -> # assertions contain "i_data_a"              = 195
44      -> # assertions contain "i_data_b"              = 122
45      -> # assertions contain "mod_reg"               = 0
46      -> # assertions contain "temp_multiply"         = 0
47      -> # assertions contain "i_inst"                = 40
48      -> # assertions contain "i_valid"               = 1
49      -> # assertions contain "o_data"                = 255
50      -> # assertions contain "o_data_w"              = 44
51      -> # assertions contain "o_valid"               = 1
```

```
53  ============================================================
54  ** Factor P Info (Parent dependency) **
55  ============================================================
56  -> The list of hierarchical depth for each cared signal
57      -> depths of "o_valid"                = 0 (not in suspect control path)
58      -> depths of "i_valid"                = 3
59      -> depths of "i_inst"                 = 2
60      -> depths of "o_data_w"               = 1
61      -> depths of "temp_multiply"          = 0 (not in suspect control path)
62      -> depths of "mod_reg"                = 0 (not in suspect control path)
63      -> depths of "i_data_b"               = 1
64      -> depths of "o_data"                 = 0 (not in suspect control path)
65      -> depths of "DATA_W"                 = 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
66      -> depths of "i_data_a"               = 1
67
68  ============================================================
69  ** Factor D Info (Distribution) **
70  ============================================================
71  -> # total control path = 16
72      -> # control path contain "o_valid"           = 1
73      -> # control path contain "o_data_w"          = 5
74      -> # control path contain "o_data"            = 1
75      -> # control path contain "i_valid"           = 3
76      -> # control path contain "i_inst"            = 3
77      -> # control path contain "temp_multiply"     = 7
78      -> # control path contain "mod_reg"           = 2
79      -> # control path contain "i_data_b"          = 6
80      -> # control path contain "i_data_a"          = 8
```

Figure 7.5: The example of the information used to calculate the score of hint signals provided to designers

In summary, our tool significantly narrows the search space by focusing on the crucial control paths, guiding designers to prioritize their debugging efforts. This approach drastically reduces the burden on designers, minimizing the need to observe testbench results, inspect dense waveforms, and employ other techniques to locate the source of the bug.


doi:10.6342/NTU202403664

# Chapter 8 Experimental Results

In this chapter, we present the experimental results to evaluate the effectiveness and efficiency of our bug localization tool. We begin with a brief overview of a blind test case study, comparing the debugging process and effort required using our tool versus a traditional designer-led approach (Section 8.1). Following this, we demonstrate the tool's generalization and applicability by showcasing additional demo cases (Section 8.2). Finally, we provide a detailed analysis of the results, including metrics such as time and memory usage, the number of assertions generated and filtered, and the identification of key signals in the debugging process (Section 8.3).

## 8.1 A Brief Overview of the Blind Test Case

In this section, we will use a DUV to illustrate the typical debugging process a designer follows without the assistance of our tool. We will then demonstrate how our tool can enhance this process. The outline is as follows: first, we introduce the specification and bug of the DUV; next, we detail the manual debugging steps a designer would take; finally, we show how our tool guides the designer, highlighting the advantages it offers

**Design: Vending Machine**


doi:10.6342/NTU202403664

A vending machine, depicted in Fig. 8.1, is designed to sell items to buyers. The process involves a buyer selecting an item and inserting coins into the machine. The vending machine should then dispense the selected item and, if the amount inserted exceeds the item's price, return the appropriate change to the buyer.

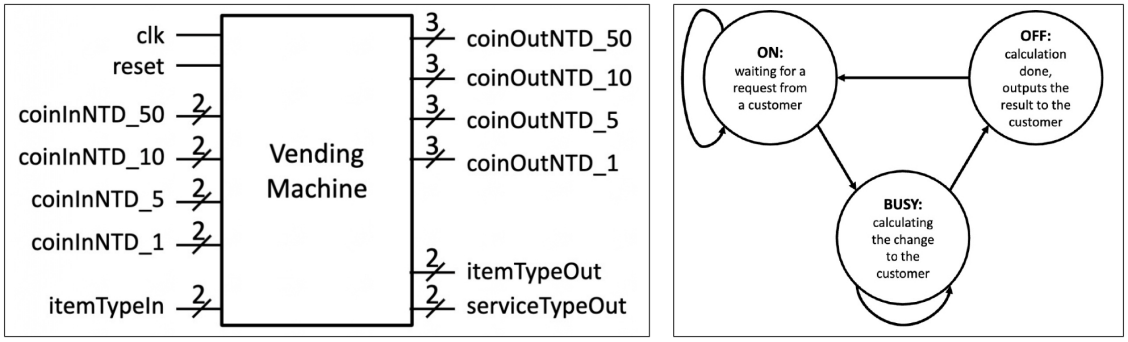



Figure 8.1: Spec of Vending Machine

**Bug: Initial Change Calculation Error—Logical Flaw in Determination Process**

Initially, when the buyer inserts coins to purchase an item, the vending machine determines whether the total coin value provided by the buyer is sufficient to cover the cost of the item. If the amount is insufficient, the vending machine should return nothing and reset. However, as shown in Fig. 8.3 (b), the DUV has a logical flaw: the machine incorrectly returns nothing and resets when "input value $\leq$ item cost" instead of the correct condition "input value $<$ item cost".

**Debugging Process of a Designer**

In this scenario, Jimmy Yang, a junior in our lab with experience in designing and debugging Verilog projects, attempts to debug the DUV. Fig. 8.2 illustrates his debugging


doi:10.6342/NTU202403664

process. Given the DUV, the reference model written in Python based on the spec, the testbench to simulate the DUV, and both pass and fail patterns, Jimmy follows these steps:

1. **Initial Analysis**: Jimmy checks the debugging messages printed by the testbench to identify which patterns begin to fail, the input pattern at the bug-triggered cycle, the output result of the DUV, the golden result, and the preceding patterns to understand the correlation.

2. **Speculation**: Using the patterns and output results, Jimmy speculates about the incorrect behavior. For example, when a buyer pays two dollars for an item that costs two dollars, the golden result should be to dispense the item without change, but the DUV outputs no item. He suspects all RTL code blocks related to item output and change calculation, which encompasses a significant portion of the RTL code.

3. **Waveform Analysis**: To avoid interference from other RTL code blocks, Jimmy delves into the waveform of the suspected RTL blocks in FSM order, examining the behavior of each internal signal related to change and item output.

4. **RTL Code Examination**: After identifying signals with unexpected behavior in the waveform, Jimmy checks the corresponding RTL code for logic errors and attempts to fix the suspected part. He then simulates the failed patterns again to verify if the issue is resolved. If not, he repeats the process for other suspected RTL code blocks.

This debugging process takes Jimmy over half an hour, involving trial and error on each suspected RTL code block and speculating on waveform behavior.

**Guidance by Our Tool—Configuration Setting and Bug Localization Result**


doi:10.6342/NTU202403664

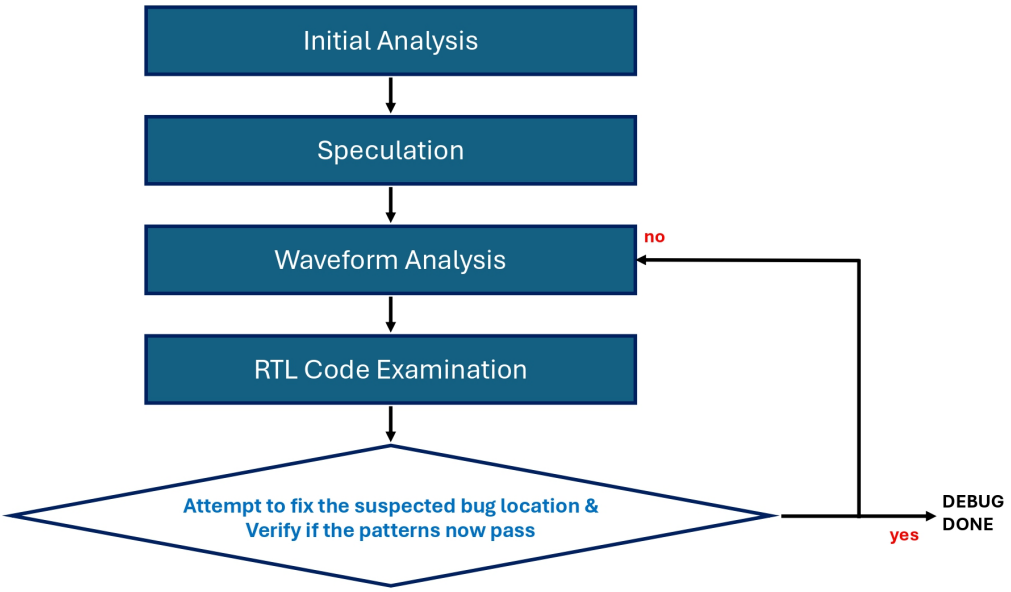



Figure 8.2: The debugging process from the designer's perspective

After Jimmy successfully identified the bug, we demonstrated our tool for him. We introduced the configuration file setting, which Jimmy set up himself in just a few minutes, as shown in Fig. 8.3 (a). By executing our tool, we identified the most suspected hint signal "inputValue" within 10+ minutes, with 96% of the time spent on Synopsys VC Formal. The control path containing this hint signal was ranked highest and was one of Jimmy's initially suspected RTL code blocks related to change and item output, shown in Fig. 8.3 (b).

With this guidance, Jimmy can skip most of the debugging time spent on observing waveforms and speculating signal behavior in unrelated RTL code blocks. Instead, he can focus directly on the suggested RTL code block and locate the bug in just one iteration, as illustrated in Fig. 8.2.

This case demonstrates that our tool effectively reduces the number of irrelevant parts considered by designers, directing them to focus on the correct debugging path. This drastically reduces debugging time and alleviates the burden as the scale and complexity


doi:10.6342/NTU202403664

of the DUV grows.


(a) (b)

Figure 8.3: (a) Configuration setting, and (b) final bug localization result of Vending Machine

## 8.2 A Brief Overview of the Demo Cases

### Design 1: Arbiter

An arbiter (Fig. 8.4) is designed to manage the allocation of a shared resource among multiple systems. When a system wants to occupy the resource, it sends a request to the arbiter. Based on its current state and the availability of the resource, the arbiter decides whether to grant the request. According to the specification, the arbiter can only grant a system's request if no other system is currently using the resource, ensuring fair and orderly access to the shared resource.

#### Bug: Resource Preemption in Multi-System Environments

A critical bug was identified in the DUV where the arbiter improperly grants access to "system 1" regardless of whether other systems are already using the resource (Fig. 8.5


doi:10.6342/NTU202403664

(b)). According to the specification, the arbiter should only grant a request if no other system is occupying the resource. However, in the DUV, the arbiter grants "system 1" access whenever it sends a request, leading to resource conflicts and interruptions.

**Configuration Setting and Bug Localization Result**

The configuration setting provided by designers in our tool is shown in Fig. 8.5 (a). Using this configuration, our tool automatically identifies the most suspected signal causing the bug. The bug localization result, displayed in Fig. 8.5 (b), highlights the *state* signal as the primary suspect. Designers can then focus on the control path involving this signal to quickly identify and rectify the bug, streamlining the debugging process significantly.

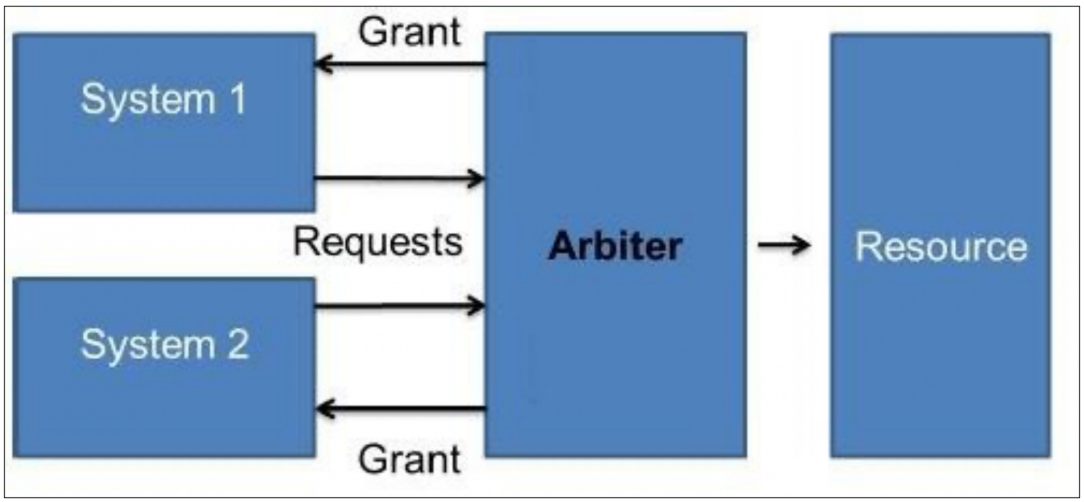



Figure 8.4: Spec of Arbiter

## Design 2: ALU

An Arithmetic Logic Unit (ALU) performs various operations on input data based on a given instruction set. In this DUV, the ALU supports 8 distinct operations as illustrated in Fig. 8.6. These operations range from basic arithmetic to more complex logical functions,


doi:10.6342/NTU202403664

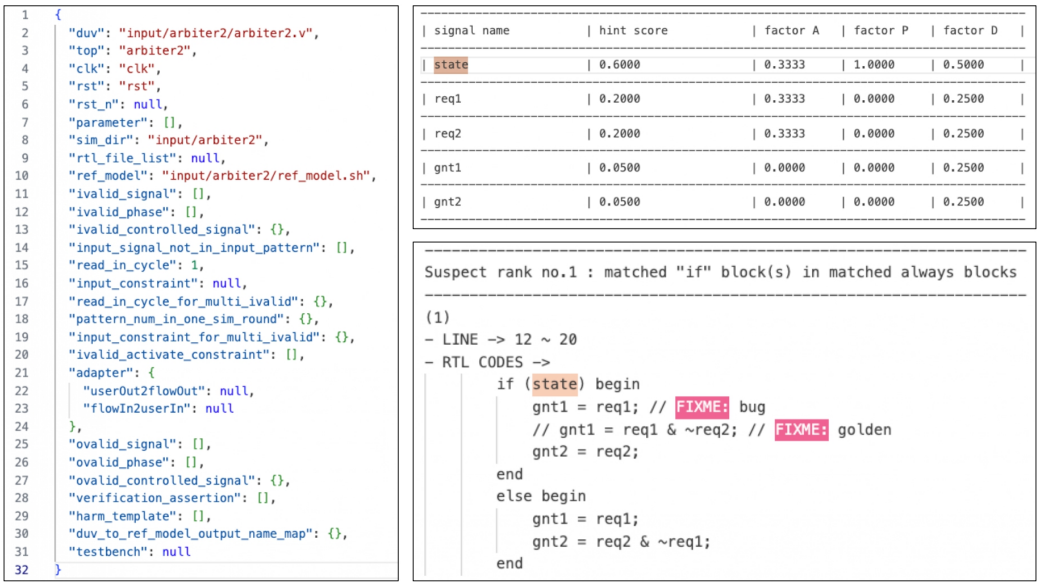



Figure 8.5: (a) Configuration setting, and (b) final bug localization result of Arbiter

controlled by specific instructions.

**Bug: Incorrect Application of Arithmetic and Logical Shift Operations**

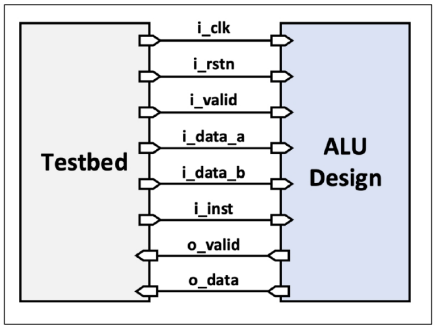
According to the specification, instruction "101" in the DUV is responsible for performing a piecewise linear approximation of the Sigmoid function. One of the formulas states that when $i\_data\_b$ is between $-2$ and 2, the approximated Sigmoid result should be "$0.25 * i\_data\_a + 0.5$". However, $i\_data\_a$ is a signed value, and the bug arises because the DUV incorrectly uses a logical right shift for the "$0.25 * i\_data\_a$" operation (Fig. 8.7 (b)), which pads the Most Significant Bit (MSB) with zero instead of using an arithmetic right shift that pads the MSB with the sign bit. This error leads to incorrect results when $i\_data\_a$ is negative.

**Configuration Setting and Bug Localization Result**

Fig. 8.7 (a) shows the configuration settings provided by designers in our tool, which, in this scenario, includes consideration of the DUV's input valid signal and defined param-

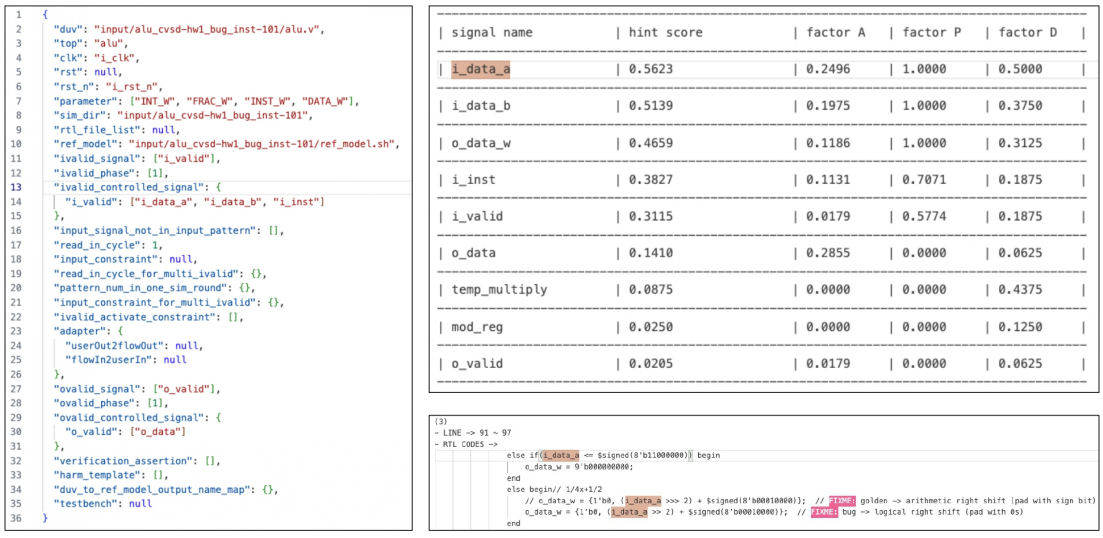

doi:10.6342/NTU202403664

eters. After applying our tool, the bug localization result is displayed in Fig. 8.7 (b). The tool highlights $i\_data\_a$ as the most suspected signal, guiding designers to examine the control path containing this signal. This targeted approach enables designers to quickly identify and correct the bug.




| i_inst [2:0] | Operation | Description |
|---|---|---|
| 000 | Signed Addition | o_data = i_data_a + i_data_b |
| 001 | Signed Subtraction | o_data = i_data_a - i_data_b |
| 010 | Signed Multiplication | o_data = i_data_a * i_data_b |
| 011 | NAND | o_data = ( i_data_a · i_data_b )' |
| 100 | XNOR | o_data =( i_data_a ⊕ i_data_b)' |
| 101 | Sigmoid | o_data = $\sigma$( i_data_a ) |
| 110 | Right Circular Shift | See 111_HW1_note.pdf |
| 111 | Min | o_data = min ( i_data_a, i_data_b ) If i_data_a = i_data_b, o_data = i_data_a |

Figure 8.6: Spec of ALU




(a) (b)

Figure 8.7: (a) Configuration setting, and (b) final bug localization result of ALU

## Design 3: Huffman Encoder



doi:10.6342/NTU202403664

Huffman coding (Fig. 8.8) is a technique used in lossless data compression, such as storing information about the number of occurrences of each gray level in an image in this case. This DUV is designed to implement the Huffman encoding algorithm, which outputs encoded data for each number and a "mask" to indicate the valid bits of the encoded result. The "mask" ensures that bits corresponding to the valid length of the encoded code are set to 1.

**Bug: Variable Name Mismatch—"$code\_length[0]$" versus "$code[0]$"**

According to the specification, the DUV should output both the encoding result for each number and a "mask" to indicate which bits are valid in the encoding result. The encoding code's bit length is stored in $code\_length$ array, while the encoding itself is stored in $code$ array. The bug arises because of a typo where $code$ is mistakenly used instead of $code\_length$ to indicate the bit length of the encoding code (Fig. 8.9 (b)). This mistake causes the "mask" to incorrectly reflect the valid bits, leading to erroneous output.

**Configuration Setting and Bug Localization Result**

Fig. 8.9 (a) illustrates the configuration settings provided by designers in our tool. In this scenario, the tool accounts for the DUV with multiple output valid signals controlling different output results. It also adapts to situations where the input pattern format and output result differ between the tool and the designer's specification by using adapters. Additionally, the tool handles cases where designers have specified input constraints in the specification. To address the instability of the SDM, the tool provides output signal matching information between the DUV and the reference model.

After applying our tool, the bug localization result is displayed in Fig. 8.9 (b). The tool highlights $current\_state$ as the most suspected signal, guiding designers to examine


doi:10.6342/NTU202403664

the control path containing this signal. This targeted approach allows designers to quickly identify and correct the variable name mismatch bug.

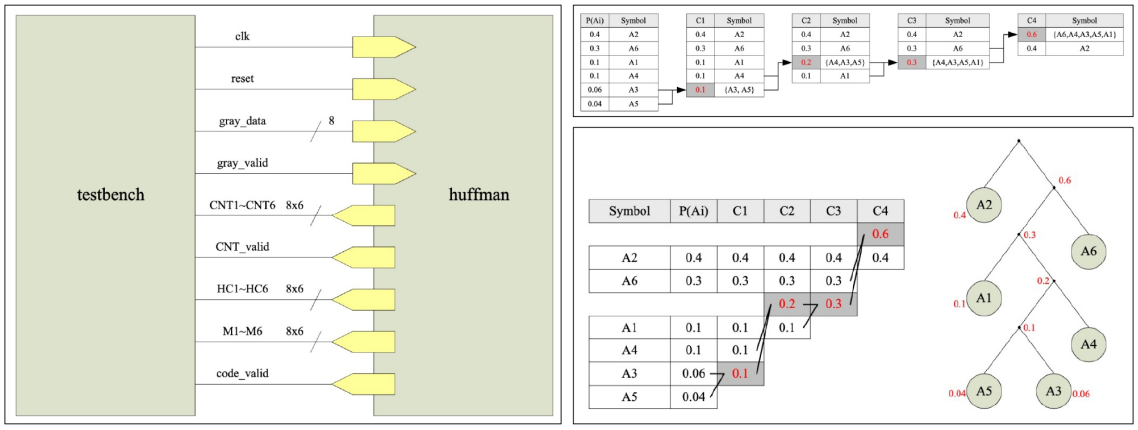



Figure 8.8: Spec of Huffman Encoder

## Design 4: GSIM

The Gauss-Seidel Iteration Machine (GSIM) is designed to solve a system of multivariate linear equations through iterative application of the formula (Fig. 8.10). The solution converges over multiple iterations.

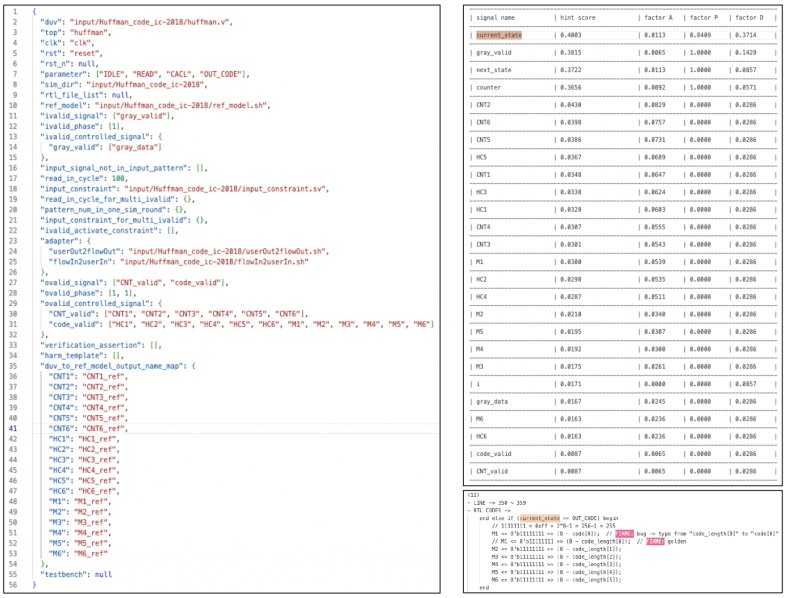
### Bug: Unintended Sign Bit Omission Due to Typographical Error

In the calculation process, a buffer is used to store temporary values and needs to store 16 bits using the notation [+:16]. However, a typographical error results in it being written as [+:15], which unintentionally excludes the signed bit and can cause errors when handling negative values (Fig. 8.11 (b)).

### Configuration Setting and Bug Localization Result

Fig. 8.11 (a) shows the configuration settings provided by designers in our tool. In


doi:10.6342/NTU202403664



(a) (b)

Figure 8.9: (a) Configuration setting, and (b) final bug localization result of Huffman Encoder

this scenario, the tool considers the DUV with multiple input valid signals controlling different input data. After applying our tool, the bug localization result is shown in Fig. 8.11 (b). Designers can start the debugging process based on our hint from the most suspected signal, $i$. They can quickly determine that this signal is insignificant and filter it out due to only few data assignment within its control path. By delving into the control path containing the second most suspected signal, $col\_cnt\_r$, designers can quickly identify the bug.

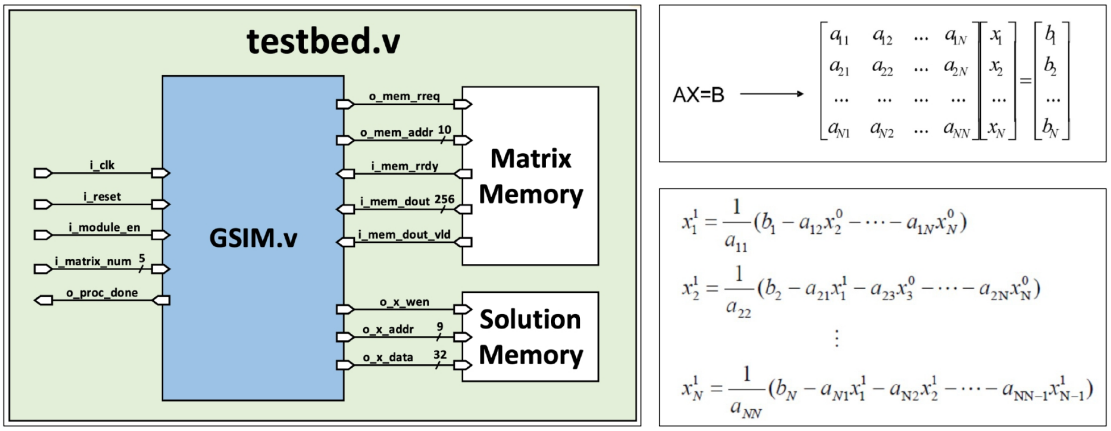
## Design 5: Convolution Engine

The DUV is designed to perform convolution on a 2048-pixel image using a defined kernel, as illustrated in Fig. 8.12. The DUV includes eight operations, such as shifting the origin position and changing the channel depth to display pixels or perform convolution.

**Bug: Origin Shift Error Violating Boundary Conditions**

In the specification, the display region is a $2 * 2$ window, with the top-left corner


doi:10.6342/NTU202403664




$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \dots & \dots & \dots & \dots \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_N \end{bmatrix}$$

$$x_1^1 = \frac{1}{a_{11}}(b_1 - a_{12}x_2^0 - \cdots - a_{1N}x_N^0)$$

$$x_2^1 = \frac{1}{a_{22}}(b_2 - a_{21}x_1^1 - a_{23}x_3^0 - \cdots - a_{2N}x_N^0)$$

$$\vdots$$

$$x_N^1 = \frac{1}{a_{NN}}(b_N - a_{N1}x_1^1 - a_{N2}x_2^1 - \cdots - a_{NN-1}x_{N-1}^1)$$

Figure 8.10: Spec of GSIM

```
{
    "duv": "input/GSIM_cvsd-final/GSIM.v",
    "top": "GSIM",
    "clk": "i_clk",
    "rst": "i_reset",
    "rst_n": null,
    "parameter": [],
    "sim_dir": "input/GSIM_cvsd-final",
    "rtl_file_list": null,
    "ref_model": "input/GSIM_cvsd-final/ref_model.sh",
    "ivalid_signal": ["i_mem_dout_vld"],
    "ivalid_phase": [1],
    "ivalid_controlled_signal": {
        "i_mem_dout_vld": ["i_mem_dout"]
    },
    "input_signal_not_in_input_pattern": ["i_module_en", "i_matrix_num", "i_mem_rrdy"],
    "read_in_cycle": 17,
    "input_constraint": null,
    "read_in_cycle_for_multi_ivalid": {},
    "pattern_num_in_one_sim_round": {},
    "input_constraint_for_multi_ivalid": {},
    "ivalid_activate_constraint": [],
    "adapter": {
        "flowIn2userIn": "input/GSIM_cvsd-final/flowIn2userIn.sh"
    },
    "ovalid_signal": ["o_x_wen"],
    "ovalid_phase": [1],
    "ovalid_controlled_signal": {
        "o_x_wen": ["o_x_data"]
    },
    "verification_assertion": [],
    "harm_template": [],
    "duv_to_ref_model_output_name_map": {},
    "testbench": "input/GSIM_cvsd-final/testbench.sh"
}
```

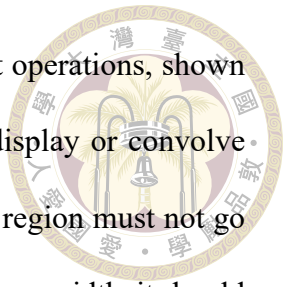
| signal name | hint score | factor A | factor P | factor D |
|---|---|---|---|---|
| i | 0.4373 | 0.0246 | 1.0000 | 0.3810 |
| col_cnt_r | 0.4135 | 0.0434 | 0.8409 | 0.3651 |
| i_module_en | 0.4099 | 0.0485 | 1.0000 | 0.1905 |
| i_mem_dout_vld | 0.4084 | 0.0402 | 0.7418 | 0.4444 |
| eq_r | 0.4056 | 0.0811 | 1.0000 | 0.0952 |
| mat_cnt_r | 0.4037 | 0.0700 | 1.0000 | 0.1111 |
| i_mem_rrdy | 0.4034 | 0.0410 | 1.0000 | 0.1746 |
| i_matrix_num | 0.3955 | 0.0164 | 1.0000 | 0.1905 |
| iter_cnt_r | 0.3632 | 0.0376 | 0.8409 | 0.1270 |
| earlyout_r | 0.3463 | 0.0000 | 0.8409 | 0.1270 |
| state_r | 0.3124 | 0.0410 | 0.5888 | 0.1270 |
| o_mem_addr | 0.1067 | 0.2019 | 0.0000 | 0.0794 |
| o_x_addr | 0.0880 | 0.1602 | 0.0000 | 0.0794 |
| o_x_data | 0.0430 | 0.0602 | 0.0000 | 0.0794 |
| o_proc_done | 0.0417 | 0.0574 | 0.0000 | 0.0794 |
| i_mem_dout | 0.0371 | 0.0471 | 0.0000 | 0.0794 |
| o_x_wen | 0.0292 | 0.0296 | 0.0000 | 0.0794 |
| o_mem_rreq | 0.0159 | 0.0000 | 0.0000 | 0.0794 |

(a) (b)

Figure 8.11: (a) Configuration setting, and (b) final bug localization result of GSIM



doi:10.6342/NTU202403664



defined as the “origin” (highlighted in yellow in Fig. 8.12). Different operations, shown in the left figure of Fig. 8.12, allow shifting the origin position to display or convolve different pixel windows. The specification mandates that the display region must not go out of bounds. This means that if the origin is at the boundary of the image width, it should not continue shifting right even if the operation is “right shift origin”. However, the DUV fails to adhere to this boundary rule (Fig. 8.13 (b)).

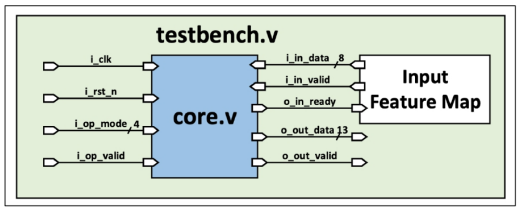
**Configuration Setting and Bug Localization Result**

In Fig. 8.13 (a), designers provide the configuration settings for our tool. In this scenario, the DUV interacts with memory, such as SRAM, and includes controlling signals with constraints that need to be satisfied between each other. We also demonstrate the verification assertions that designers can provide to enhance the bug localization process, particularly focusing on conditions in boundary conditions as regulated in the spec. After applying our tool, the bug localization result is shown in the right figure. Designers can use our hint to start the debugging process from the most suspected signal, “origin_r”. By delving into the control path containing this signal, designers can quickly identify the bug.

## 8.3 Results

Our experiments were conducted on a Linux server featuring dual Intel® Xeon® E5-2630 v4 processors (10 cores each, 2.2GHz), totaling 40 CPU cores, with 125GB RAM. The tools employed were as follows: SDM for output signal matching, HARM for automatic assertion mining, Synopsys VC Formal for formal verification, Icarus Verilog and Synopsys VCS for RTL simulation, and Pyverilog for parsing RTL code structure. Our algorithm was implemented in C++ and compiled using G++ with -O3 optimization.


doi:10.6342/NTU202403664




| i_op_mode | Meaning |
|---|---|
| 4'b0000 | Input feature map loading |
| 4'b0001 | Origin right shift |
| 4'b0010 | Origin left shift |
| 4'b0011 | Origin up shift |
| 4'b0100 | Origin down shift |
| 4'b0101 | Reduce the number of channels for convolution |
| 4'b0110 | Increase the number of channels for convolution |
| 4'b0111 | Perform convolution in the display region |
| 4'b1000 | Output the pixels in the display region |

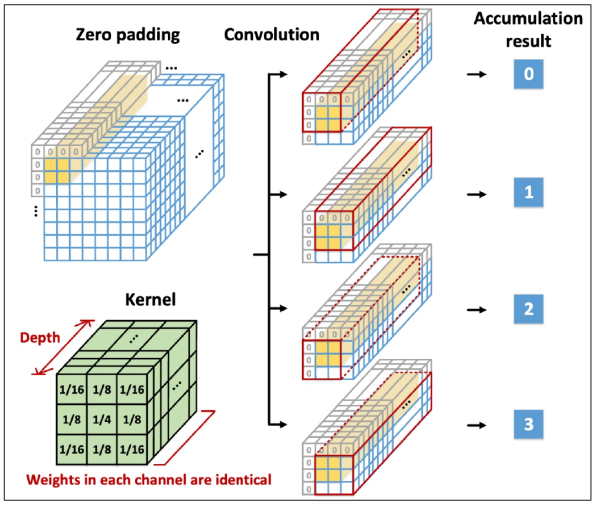
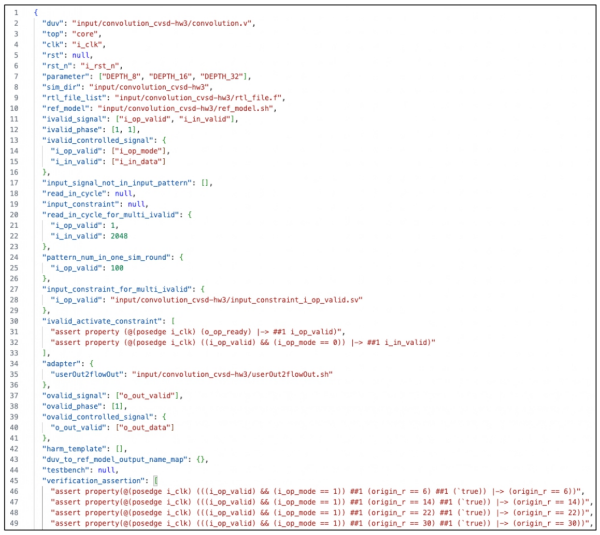



Figure 8.12: Spec of Convolution Engine



```
{
  "duv": "input/convolution_cvsd-hw3/convolution.v",
  "top": "core",
  "clk": "i_clk",
  "rst": null,
  "rst_n": "i_rst_n",
  "parameter": ["DEPTH_0", "DEPTH_16", "DEPTH_32"],
  "sim_dir": "input/convolution_cvsd-hw3",
  "rtl_file_list": "input/convolution_cvsd-hw3/rtl_file.f",
  "ref_model": "input/convolution_cvsd-hw3/ref_model.sh",
  "ivalid_signal": ["i_op_valid", "i_in_valid"],
  "ivalid_phase": [1, 1],
  "ivalid_controlled_signal": {
    "i_op_valid": ["i_op_mode"],
    "i_in_valid": ["i_in_data"]
  },
  "input_signal_not_in_input_pattern": [],
  "read_in_cycle": null,
  "input_constraint": null,
  "read_in_cycle_for_multi_ivalid": {
    "i_op_valid": 1,
    "i_in_valid": 2048
  },
  "pattern_num_in_one_sim_round": {
    "i_op_valid": 100
  },
  "input_constraint_for_multi_ivalid": {
    "i_op_valid": "input/convolution_cvsd-hw3/input_constraint_i_op_valid.sv"
  },
  "ivalid_activate_constraint": [
    "assert property (@(posedge i_clk) (o_op_ready) |-> ##1 i_op_valid)",
    "assert property (@(posedge i_clk) ((i_op_valid) && (i_op_mode == 0)) |-> ##1 i_in_valid)"
  ],
  "adapter": {
    "userOut2flowOut": "input/convolution_cvsd-hw3/userOut2flowOut.sh"
  },
  "ovalid_signal": ["o_out_valid"],
  "ovalid_phase": [1],
  "ovalid_controlled_signal": {
    "o_out_valid": ["o_out_data"]
  },
  "harm_template": [],
  "duv_to_ref_model_output_name_map": {},
  "testbench": null,
  "verification_assertion": [
    "assert property(@(posedge i_clk) (((i_op_valid) && (i_op_mode == 1)) ##1 (origin_r == 6) ##1 (`true)) |-> (origin_r == 6))",
    "assert property(@(posedge i_clk) (((i_op_valid) && (i_op_mode == 1)) ##1 (origin_r == 14) ##1 (`true)) |-> (origin_r == 14))",
    "assert property(@(posedge i_clk) (((i_op_valid) && (i_op_mode == 1)) ##1 (origin_r == 22) ##1 (`true)) |-> (origin_r == 22))",
    "assert property(@(posedge i_clk) (((i_op_valid) && (i_op_mode == 1)) ##1 (origin_r == 30) ##1 (`true)) |-> (origin_r == 30))",
```

(a)

| signal name | hint score | factor A | factor P | factor D |
|---|---|---|---|---|
| origin_r | 0.4785 | 0.1829 | 1.0000 | 0.2308 |
| sram_CEN_r | 0.4167 | 0.0000 | 1.0000 | 0.3333 |
| depth_r | 0.4064 | 0.0000 | 1.0000 | 0.2821 |
| i_in_valid | 0.4029 | 0.0151 | 1.0000 | 0.2308 |
| i_op_valid | 0.4029 | 0.0151 | 1.0000 | 0.2308 |
| display_depth_w | 0.3876 | 0.0151 | 1.0000 | 0.1538 |
| display_depth_r | 0.3859 | 0.0000 | 1.0000 | 0.1795 |
| sram_A_r | 0.3859 | 0.0000 | 1.0000 | 0.1795 |
| first_data_r | 0.3808 | 0.0000 | 1.0000 | 0.1538 |
| op_mode_r | 0.3716 | 0.0213 | 0.8409 | 0.2051 |
| state_r | 0.3374 | 0.0370 | 0.6389 | 0.2051 |
| i_in_data | 0.1774 | 0.3486 | 0.0000 | 0.1026 |
| i_op_mode | 0.1082 | 0.1950 | 0.0000 | 0.1026 |
| o_out_data | 0.0722 | 0.1149 | 0.0000 | 0.1026 |
| o_op_ready | 0.0357 | 0.0337 | 0.0000 | 0.1026 |
| o_out_valid | 0.0301 | 0.0213 | 0.0000 | 0.1026 |
| raster_scan_state_r | 0.0256 | 0.0000 | 0.0000 | 0.1282 |
| conv_state_r | 0.0256 | 0.0000 | 0.0000 | 0.1282 |
| o_in_ready | 0.0205 | 0.0000 | 0.0000 | 0.1026 |

```
(2)
- LINE --> 235 ~ 236
- RTL CODES -->
                if(origin_r[2:0] != 3'd7) origin_w = origin_w + 1'd1;    // FROM: bug, out of boundary
                else                      origin_w = origin_w;
```

(b)

Figure 8.13: (a) Configuration setting, and (b) final bug localization result of Convolution Engine



doi:10.6342/NTU202403664

The following tables present the experimental results for each DUV demonstrated in Section 8.1 and Section 8.2. Table. 8.1 and Table. 8.2 illustrate the complexity of the bug localization task, including the scale, number of control paths, and signals that designers need to filter to identify the bug. Our filtering algorithm effectively reduces this complexity by ignoring irrelevant signals and focusing on hint signals. By applying our metrics to sort these hint signals, we provide designers with guidance on the top suspected control paths containing the hint signals. Our approach consistently locates the bug either in the first or, in the worst case, the second priority of suspected hint signals. Detailed bug localization results are presented in Section 8.1 and Section 8.2.

Table 8.1: The Ranking of Hint Signals of DUVs in Section 8.1 and 8.2

| DUV | Hint Signals | | | |
|---|---|---|---|---|
| - | # Control Path | # Total Sigs | # Hint Sigs | Top-X Finding the Bug |
| Vending Machine | 12 | 26 | 13 | 1 |
| Arbiter | 4 | 7 | 5 | 1 |
| ALU | 16 | 14 | 9 | 1 |
| Huffman Encoder | 35 | 29 | 26 | 1 |
| GSIM | 63 | 39 | 18 | 2 |
| Convolution Engine | 39 | 55 | 19 | 1 |

Table 8.2: Design scale of DUVs in Section 8.1 and 8.2

| DUV | Scale | | | |
|---|---|---|---|---|
| - | # RTL lines (DUV) | # AND gate (AIG) | # Flip-flop (AIG) | Size |
| Vending Machine | 176 | 528 | 25 | middle |
| Arbiter | 22 | 10 | 1 | small |
| ALU | 133 | 40528 | 44 | middle |
| Huffman Encoder | 471 | 5934 | 338 | middle |
| GSIM | 918 | 132803 | 1341 | large |
| Convolution Engine | 501 | 188081 | 2702 | large |

Table. 8.3 illustrates the filtering process for assertions, detailing the number of assertions at different stages. Initially, we use HARM to automatically generate assertion candidates, as shown in the first column of *# Assertions* section. We also consider designer-provided assertions to enhance the bug localization process, though no user-provided assertions were included in our experiments. In the first filtering process, we use VC Formal


doi:10.6342/NTU202403664

to prove the correctness of assertion candidates, retaining only the valid assertions for the second filtering process. After applying our algorithm, the final count of valid and suspected assertion candidates is shown in the last column, demonstrating our ability to focus on the most relevant information to the bug. Table. 8.4 presents the computational time and memory utilization for the third-party tools and our algorithm, respectively.

In Table. 8.4, we observe a runtime gap between the first three cases and the last three. This is because the last three cases involve designer-provided input constraints, requiring the use of the commercial tool Synopsys VCS for each round of constrained random simulation. This significantly increases the computational time. Addressing this limitation is part of our future work, as outlined in Chapter 9.

Table 8.3: The Number of Assertions of DUVs in Section 8.1 and 8.2

| DUV | # Assertions | | | |
|---|---|---|---|---|
| - | HARM Mined | User Provided | VCF Proven | Our Final Candidates |
| Vending Machine | 7758 | 0 | 466 | 116 |
| Arbiter | 93 | 0 | 6 | 2 |
| ALU | 22882 | 0 | 1235 | 658 |
| Huffman Encoder | 56173 | 0 | 1468 | 1273 |
| GSIM | 22504 | 0 | 1939 | 1214 |
| Convolution Engine | 21677 | 528 | 1965 | 767 |

Table 8.4: Computational Time and Memory Utilization of DUVs in Section 8.1 and 8.2

| DUV | Time (s) | | | | Peak Memory (MB) | |
|---|---|---|---|---|---|---|
| - | HARM | VC Formal | Our Tool | Total | VC Formal | Our Tool |
| Vending Machine | 5.94 | 1003.59 | 30.64 | 1040.17 | 2929 | 20.90 |
| Arbiter | 0.07 | 26.67 | 1.85 | 28.59 | 1246 | 5.08 |
| ALU | 44.30 | 1603.27 | 77.95 | 1725.52 | 6085 | 133.72 |
| Huffman Encoder | 894.50 | 9737.12 | 6624.34 | 17255.96 | 11773 | 377.11 |
| GSIM | 346.60 | 4771.13 | 2346.02 | 7463.75 | 7972 | 2479.38 |
| Convolution Engine | 78.33 | 4298.08 | 5694.61 | 10071.02 | 9927 | 12234.20 |

To the best of our knowledge, there are few end-to-end bug localization tools similar to ours, and those that exist do not open-source their tools or the designs and bugs they use. Most other works focus only on parts of the phases we cover in Chapters 5 to 7. Consequently, it is difficult to use a benchmark to demonstrate the accuracy of our bug


doi:10.6342/NTU202403664

localization. Therefore, in the following paragraph, we will reference claims from related bug localization papers to show the relative accuracy of our approach.

First, we will compare with [1]. Our bug localization process, detailed in Chapter 7, builds upon this paper's method of categorizing the suspected control path into four "suspected bins", which represent debugging priorities. However, this paper only narrows down the bug location to a bin, which may still contain many candidate control paths. Our work enhances this by further pinpointing the most suspected control path within each bin by ranking the controlling signals by suspicion, which increases accuracy. As shown in Table. 8.1, our experimental results can identify the correct bug position's control path signal in the first or second rank.

While the other work [26] focuses on designs containing instructions, such as ALU or MIPS. Their approach involves training a multi-class classifier to locate bugs in potentially incorrect instruction source code, using the "F1 score" as the metric to evaluate the ML algorithm's performance. In their experimental results, the F1 scores for each case are less than 50 percent accurate. While this paper explores using ML techniques for bug localization, their method is less general and accurate compared to ours.


doi:10.6342/NTU202403664

# Chapter 9 Conclusion and Future Work

As we conclude this comprehensive exploration into bug localization within DUV, we reflect on the significant advancements made and outline the potential directions for future research and development. This chapter is divided into two sections: Section 9.1 summarizes our key contributions and findings, while Section 9.2 discusses promising areas for further investigation and enhancement of our methodology.

## 9.1 Conclusion

In our work, we aim to assist designers in bug localization within a DUV. Given a DUV and its configuration settings, our tool generates numerous assertion candidates and filters them to retain only the valid and suspected assertions through constrained random simulation. These suspected assertions are then used to pinpoint potential bug zones within the DUV's code.

Our primary contribution is the integration of various third-party tools, each specializing in different tasks such as formal verification and assertion generation, into a comprehensive flow for bug localization. Additionally, we introduce methods for constrained


doi:10.6342/NTU202403664

pattern generation and assertion validation to effectively filter valid and suspected assertion candidates, focusing on identifying the bug. This approach significantly reduces the effort required by designers to locate potential bugs within dense waveforms and numerous signals, thereby shortening debugging time.

## 9.2 Future Work

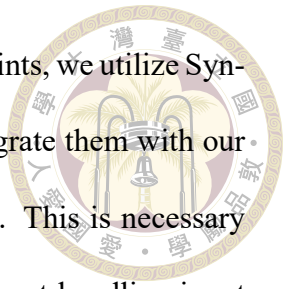
As we look to the future, several potential improvements could make our tool more generalized and comprehensive in locating bugs.

- **Enhancing the Stability of SDM**: The open-source third-party tool SDM currently faces instability issues when matching output signals between the DUV and the reference model. To address this, we can improve SDM's algorithm or provide additional useful information to increase its accuracy. This enhancement could allow us to eliminate the need for the *-duv_to_ref_model_output_name_map* parameter in the configuration file, which is currently used when SDM fails to match output signals correctly.

- **Improving Assertion Quality in HARM**: The quality of assertions mined by HARM, another open-source third-party tool, can be refined. HARM's performance heavily depends on the quality of the given simulation results and the assertion mining template. By improving HARM's algorithm and tuning the parameters in the assertion mining template, we can generate higher-quality assertions for more effective bug localization, regardless of the quality of the given simulation results.

- **Reducing Computational Time by Minimizing Synopsys VCS Calls**: Currently,


doi:10.6342/NTU202403664



when a DUV is configured with designer-specified input constraints, we utilize Synopsys VCS to read these constraints from an SVA file and integrate them with our algorithm's soft constraints for constrained random simulation. This is necessary because open-source simulators like Icarus Verilog do not support handling input constraint files. However, our algorithm requires numerous simulation runs, resulting in frequent calls to Synopsys VCS, which significantly increases computational time. For example, comparing the DUVs "Huffman Encoder" and "GSIM" in Table. 8.2 and Table. 8.4, we observe that despite the former having a smaller scale, its computational time is considerably higher. Thus, improving the handling of designer-provided input constraints can drastically reduce computational time, making the process more efficient.

- **Expanding Bug Localization to the Data Path**: Currently, our tool focuses on locating bugs in the control path. In future work, we aim to utilize more information from the simulation results to also delve into the data path. This approach would refine our bug localization process, allowing us to identify bugs with greater accuracy.

By implementing these improvements, we can enhance our tool's effectiveness and reliability, making it a more powerful resource for designers in the bug localization process.


doi:10.6342/NTU202403664

# References


[1] Binod Kumar, V. S. Vineesh, Puneet Nemade, and Masahiro Fujita. Aries: A semiformal technique for fine-grained bug localization in hardware designs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 41(12):5709–5721, 2022.

[2] Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. A survey on assertion-based hardware verification. ACM Comput. Surv., 54(11s), sep 2022.

[3] Stuart Sutherland. Who put assertions in my rtl code? and why? how rtl design engineers can benefit from the use of systemverilog assertions. https://sutherland-hdl.com/papers/2015-SNUG-SV_SVA-for-RTL-Designers_paper.pdf, 2015. SNUG Silicon Valley 2015.

[4] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In 2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010), pages 626–629, 2010.

[5] Lingyi Liu, David Sheridan, Viraj Athavale, and Shobha Vasudevan. Automatic generation of assertions from system level design using data mining. In Ninth




doi:10.6342/NTU202403664

ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011), pages 191–200, 2011.

[6] Lingyi Liu, Chen-Hsuan Lin, and Shobha Vasudevan. Word level feature discovery to enhance quality of assertion mining. In Proceedings of the International Conference on Computer-Aided Design, ICCAD '12, page 210 – 217, New York, NY, USA, 2012. Association for Computing Machinery.

[7] Alessandro Danese, Tara Ghasempouri, and Graziano Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. In 2015 Design, Automation Test in Europe Conference Exhibition (DATE), pages 67–72, 2015.

[8] Tong Zhang, Daniel Saab, and Jacob A. Abraham. Automatic assertion generation for simulation, formal verification and emulation. In 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pages 471–476, 2017.

[9] Nicola Bombieri, Riccardo Filippozzi, Graziano Pravadelli, and Francesco Stefanni. Rtl property abstraction for tlm assertion-based verification. In 2015 Design, Automation Test in Europe Conference Exhibition (DATE), pages 85–90, 2015.

[10] Tara Ghasempouri, Alessandro Danese, Graziano Pravadelli, Nicola Bombieri, and Jaan Raik. Rtl assertion mining with automated rtl-to-tlm abstraction. In 2019 Forum for Specification and Design Languages (FDL), pages 1–8, 2019.

[11] Hasini Witharana, Aruna Jayasena, Andrew Whigham, and Prabhat Mishra. Automated generation of security assertions for rtl models. J. Emerg. Technol. Comput. Syst., 19(1), jan 2023.

[12] Alessandro Danese, Nicolò Dalla Riva, and Graziano Pravadelli. A-team: Au-



doi:10.6342/NTU202403664

tomatic template-based assertion miner. In 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), pages 1–6, 2017.

[13] Samuele Germiniani and Graziano Pravadelli. Harm: A hint-based assertion miner. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 41(11):4277–4288, 2022.

[14] Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. Using llms to facilitate formal verification of rtl. https://arxiv.org/abs/2309.09437, 2023.

[15] Viraj Athavale, Sai Ma, Samuel Hertz, and Shobha Vasudevan. Code coverage of assertions using rtl source code analysis. In 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC), pages 1–6, 2014.

[16] Alessandro Danese, Francesca Filini, Tara Ghasempouri, and Graziano Pravadelli. Automatic generation and qualification of assertions on control signals: A time window-based approach. volume 483, pages 193–221, 09 2016.

[17] Tara Ghasempouri, Siavoosh Payandeh Azad, Behrad Niazmand, and Jaan Raik. An automatic approach to evaluate assertions' quality based on data-mining metrics. In 2018 IEEE International Test Conference in Asia (ITC-Asia), pages 61–66, 2018.

[18] Hui-Na Chao, Hua-Wei Li, Xiaoyu Song, Tian-Cheng Wang, and Xiao-Wei Li. Evaluating and constraining hardware assertions with absent scenarios. Journal of Computer Science and Technology, 35(5):1198–1216, 2020.

[19] Debjit Pal, Spencer Offenberger, and Shobha Vasudevan. Assertion ranking using rtl source code analysis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(8):1711–1724, 2020.



doi:10.6342/NTU202403664

[20] Mohammad Reza Heidari Iman, Jaan Raik, Maksim Jenihhin, Gert Jervan, and Tara Ghasempouri. An automated method for mining high-quality assertion sets. Microprocessors and Microsystems, 97:104773, 2023.

[21] Tai-Ying Jiang, C.-N.J. Liu, and Jing-Yang Jou. Effective error diagnosis for rtl designs in hdls. In Proceedings of the 11th Asian Test Symposium, 2002. (ATS '02)., pages 362–367, 2002.

[22] Saeed Mirzaeian, Feijun Zheng, and K.-T. Tim Cheng. Rtl error diagnosis using a word-level sat-solver. In 2008 IEEE International Test Conference, pages 1–8, 2008.

[23] Mehdi Dehbashi and Görschwin Fey. Debug automation for synchronization bugs at rtl. In 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, pages 44–49, 2014.

[24] V. S. Vineesh, Binod Kumar, Rushikesh Shinde, Neelam Sharma, Masahiro Fujita, and Virendra Singh. Enhanced design debugging with assistance from guidance-based model checking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 40(5):985–998, 2021.

[25] Neil Veira, Zissis Poulos, and Andreas Veneris. Suspect set prediction in rtl bug hunting. In 2018 Design, Automation Test in Europe Conference Exhibition (DATE), pages 1544–1549, 2018.

[26] Sanjay Rajashekar. A study on machine learning-based hardware bug localization, 2020. Available at https://oaktrust.library.tamu.edu/handle/1969.1/191832.

[27] Debjit Pal and Shobha Vasudevan. Symptomatic bug localization for functional debug of hardware designs. In 2016 29th International Conference on VLSI Design



doi:10.6342/NTU202403664

and 2016 15th International Conference on Embedded Systems (VLSID), pages 517–522, 2016.

[28] Vighnesh Iyer, Donggyu Kim, Borivoje Nikolic, and Sanjit A. Seshia. Rtl bug localization through ltl specification mining (wip). MEMOCODE '19, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Samuele Germiniani and Graziano Pravadelli. Harm: A hint-based assertion miner. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 41(11):4277–4288, 2022.

[30] Doulos. Systemverilog assertions tutorial. https://www.doulos.com/knowhow/systemverilog/systemverilog-tutorials/systemverilog-assertions-tutorial/.

[31] ChipVerify. Uvm tutorial. https://www.chipverify.com/tutorials/uvm.

[32] VLSI Verify. Uvm introduction. https://vlsiverify.com/uvm/.

[33] Chuang Bo-Han. Rtl bug localization via sequential design matching, 2023. Available at http://tdr.lib.ntu.edu.tw/jspui/handle/123456789/88888.

[34] Samuele Germiniani and Graziano Pravadelli. Harm source code (github). https://github.com/SamueleGerminiani/harm.

[35] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. volume 9040, pages 451–460, 04 2015.

[36] Shinya Takamaeda-Yamazaki. Pyverilog source code (github). https://github.com/PyHDI/Pyverilog.



doi:10.6342/NTU202403664

[37] Stephen Williams. Icarus verilog official documentation. https://steveicarus.github.io/iverilog/.

[38] Stephen Williams. Icarus verilog source code (github). https://github.com/steveicarus/iverilog.

[39] Synopsys. Vcs official website. https://www.synopsys.com/verification/simulation/vcs.html.

[40] Synopsys. Vcs official user guide documentation. http://cc.ee.ntu.edu.tw/~ric/teaching/SoC_Verification/S06/Homework/HW1/vcs.pdf.

[41] Synopsys. Vcf official website. https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html.



doi:10.6342/NTU202403664

# Appendix A — Input Pattern Format

In our tool, we define the input pattern format used to feed both the DUV and the reference model during constrained random simulation. As shown in Fig. A.1, we generalize the input signals by only considering those that carry data, excluding the clock signal, reset signal, and input valid signal specified in the configuration file. These excluded signals will not be part of the input pattern but will be used in the testbench's controlling statement. For the signals we do care about, shown on the left of Fig. A.1, we parse the declaration order written in the DUV and place them in the input pattern file from right (least significant bit) to left (most significant bit).

Designers typically may not be familiar with the input pattern format required by our tool before they need to use it. As a result, their input pattern format might differ significantly from ours. Therefore, we request designers to provide an adapter in the configuration file to convert their input pattern format to the format compatible with our tool.



doi:10.6342/NTU202403664

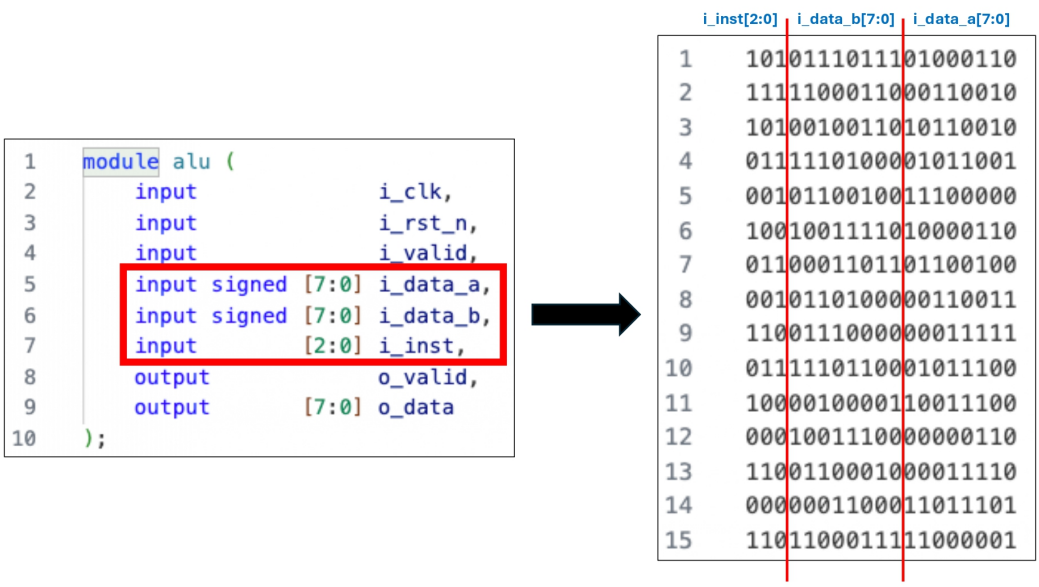



Figure A.1: The example to illustrate the concept of the input format



doi:10.6342/NTU202403664

# Appendix B — Output Result Format

In our tool, we define the format for the reference model's output results, which is necessary for the third-party tool, SDM, we utilize. According to [33], in addition to providing the DUV's pass and fail simulation results in VCD files, SDM also requires designers to supply the corresponding output result files from the reference model in CSV (Comma-Separated Values) format. These files must be generated under the same pass and fail patterns. The format is illustrated in Fig. B.2: the first row lists the output signal names from the reference model, the second row specifies the signal widths corresponding to these output signal names, and the subsequent rows present the binary string values of these output signals for each cycle.

Similarly to Appendix A, the output format used by designers may differ significantly from our tool's output format, or the reference model may not produce a CSV file at all. Therefore, we request designers to provide an adapter in the configuration file to convert their output format to one that is compatible with our tool.



doi:10.6342/NTU202403664

```
1   CNT1_ref,CNT2_ref,CNT3_ref,CNT4_ref,CNT5_ref,CNT6_ref,HC1_ref,H
2   8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8
3   00101111,00011010,00001000,00000100,00000010,00001101,00000001,
4   00110001,00001111,00001010,00000010,00001100,00001100,00000001,
5   00101001,00000101,00001100,00001000,00000100,00011110,00000001,
6   00110000,00011010,00000111,00000011,00000100,00001100,00000001,
7   00101010,00001010,00000111,00000101,00001111,00010101,00000001,
8   00101001,00001000,00001011,00001010,00000100,00011010,00000001,
9   00110001,00010101,00000110,00000010,00000110,00010000,00000001,
10  00101001,00010001,00000001,00001000,00010000,00010001,00000001,
11  00101010,00000111,00001011,00000110,00001110,00010100,00000001,
12  00110001,00001011,00000011,00000010,00000111,00011100,00000001,
13  00110000,00011001,00000001,00000011,00000001,00010110,00000001,
14  00110000,00000101,00001101,00001000,00001011,00001111,00000001,
15  00101110,00001111,00000010,00000011,00010000,00010010,00000001,
16  00101010,00011001,00000111,00000010,00001000,00010000,00000001,
17  00101011,00000111,00001010,00001001,00000110,00011001,00000001,
18  00110000,00010101,00000010,00000111,00000100,00010010,00000001,
19  00101111,00010000,00001001,00000010,00001000,00010010,00000001,
20  00101111,00010001,00000010,00000001,00001101,00010100,00000001,
```

Figure B.2: The example of the output format in a CSV file



doi:10.6342/NTU202403664