REINFORCEMENT LEARNING GUIDED INPUT GENERATION FOR FASTER ASSERTION-BASED VERIFICATION

by

Oğuz Kerem Tural

B.S., Computer Engineering, Istanbul Technical University, 2017

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2021

ABSTRACT

REINFORCEMENT LEARNING GUIDED INPUT GENERATION FOR FASTER ASSERTION-BASED VERIFICATION

Functional verification is necessary to ensure the functional correctness of digital designs. Debugging and validating design specifications during verification becomes extremely time-consuming with the increased complexity of modern register-transfer-level designs. Assertion-based verification is an effective way to reduce time spent on debugging and validation of design specifications.

In this thesis, we present a framework for assertion-based verification with a reinforcement learning-guided input generation algorithm. Our input generation algorithm uses Q-Learning to learn from the assertions and generate inputs that satisfy the given assertions faster. We have tested our framework with a 5-stage 64-bit RISC-V core and achieved significant performance improvement over random input generation.

ÖZET

DAHA HIZLI BİLDİRİM TABANLI DOĞRULAMA İÇİN DESTEKLİ ÖĞRENME GÜDÜMLÜ GİRDİ ÜRETME

İşlevsel doğrulama, dijital tasarımların işlevsel doğruluğunu sağlamak için çok önemli bir süreçtir. Modern dijital tasarımların artan karmaşıklığı ile doğrulama sırasında tasarım spesifikasyonlarının hatalarının ayıklanması ve doğrulanması son derece zaman alıcı hale gelmiştir. Bildirime dayalı doğrulama, tasarım özelliklerinin hatalarının ayıklanması ve doğrulanması için harcanan zamanı azaltmanın etkili bir yoludur.

Bu tezde, destekli öğrenme güdümlü girdi oluşturma algoritması ile bildirime dayalı doğrulama için bir çerçeve sunuyoruz. Girdi oluşturma algoritmamız, bildirimlerden öğrenmek ve verilen bildirimleri daha hızlı karşılayan girdiler oluşturmak için Q-Learning'i kullanır. Çerçevemizi 5 aşamalı 64-bit RISC-V çekirdeğiyle test ettik ve rastgele girdi üretimine göre önemli performans artışı sağladık.

TABLE OF CONTENTS

ABSTRACT . . . iii
ÖZET . . . iv
LIST OF FIGURES . . . vii
LIST OF TABLES . . . viii
LIST OF SYMBOLS . . . ix
LIST OF ACRONYMS/ABBREVIATIONS . . . x
1. INTRODUCTION . . . 1
1.1. Motivation . . . 1
1.2. Contributions . . . 4
1.3. Outline . . . 4
2. BACKGROUND . . . 5
2.1. Linear Temporal Logic . . . 5
2.1.1. Semantics of LTL . . . 5
2.2. Assertion-based Verification . . . 6
2.2.1. Properties . . . 6
2.2.2. Assertions . . . 7
2.3. Reinforcement Learning . . . 7
2.3.1. Q-Learning . . . 9
3. RELATED WORKS . . . 10
4. METHODOLOGY . . . 12
4.1. Reinforcement Learning Guided Input Generation . . . 12
4.1.1. Implementation . . . 15
5. EXPERIMENTAL RESULTS . . . 22
5.1. Experimental Setup . . . 22
5.1.1. Overview . . . 22
5.1.2. Input Specifications . . . 23
5.1.3. State Information . . . 24
5.1.4. Example . . . 24

5.1.5. Experimental Results 27
5.1.5.1. Impact of the maximum episode limit 29
5.1.5.2. Impact of the assertion complexity 30
6. CONCLUSION 32
REFERENCES 33

LIST OF FIGURES

Figure 2.1. General interaction between agent and environment. 8

Figure 4.1. Model of the simulation-based verification as a reinforcement learning process. 13

Figure 4.2. Detailed overview of the verification process of our framework. . . 14

Figure 4.3. The main algorithm of RL-Guided input generation. 17

Figure 4.4. Example state encoding table. 18

Figure 5.1. A general overview of module hierarchy. 22

LIST OF TABLES

Table 5.1. Overview of RV32IM instruction set architecture. 24

Table 5.2. Execution steps for the example run. 26

Table 5.3. List of assertions used for evaluation. 27

Table 5.4. Average Episodes Across 10 Runs. 28

Table 5.5. Average Times Across 10 Runs. 28

Table 5.6. Resulting test sequences for the assertions provided in Table 5.3 . . 29

Table 5.7. Average Episodes for Our Framework Across 10 Runs. 29

Table 5.8. Number of Failures Across 10 Runs. 29

Table 5.9. List of assertions used to test the impact of the assertion complexity. 30

Table 5.10. Resulting test sequences for the assertions provided in Table 5.9 . . 30

Table 5.11. Average Episodes Across 10 Runs. 31

Table 5.12. Average Times Across 10 Runs. 31

Table 5.13. Number of Failures Across 10 Runs. 31

LIST OF SYMBOLS

a_t	Action selected at time t
s_t	The state vector at time t
r_t	The reward given at time t
$L(s_t)$	Labels
ts_n	Test sequence
E	The maximum number of episodes
T	The maximum number of simulation steps
Q	Q-Table
ϕ	Assertion
α	Learning rate of the Q-Function
γ	Discount factor of the Q-Function
π	Policy
θ	Input specifications
Ω	The programmatic model of the DUV

LIST OF ACRONYMS/ABBREVIATIONS

LTL	Linear Temporal Logic
RTL	Register Transfer Level
DUV	Design Under Verification
RL	Reinforcement Learning
ISA	Instruction Set Architecture

1. INTRODUCTION

1.1. Motivation

Digital devices have become an essential aspect of our daily lives. Today, they replaced their analog counterparts in different areas and enabled more innovative products to come to life in various commercial, industrial, and scientific enterprises. Digital design is the process of designing digital circuits used in these devices based on a specific set of specifications [1]. Preparation of these specifications, called design specifications, is the first step of the digital design process. They define how circuits should work in a high-level language to guide engineers implementing the circuitry. After implementation, verification engineers verify that the implementation behaves as described in the design specifications. This step is called functional verification. After the functional verification step, the circuit becomes ready for mass production.

The new developments in semiconductor manufacturing technologies allowed companies to produce more complicated designs. Although this enabled development of more innovative designs, the time project teams spent on functional verification increased considerably. In recent years, project teams have spent 56% of project time verifying newly developed design intellectual properties on average, which may increase up to 80% for some projects [2]. Furthermore, time spent on debugging design in the verification process and product's time to market pressure affects the project's success. Along with the increased design complexity and diversity of modern designs, verification has become indispensable to ensure the functional correctness of those designs.

The main idea behind functional verification is finding discrepancies between design specifications and its register transfer level (RTL) design implementation. There are two main approaches to functional verification: Formal verification and simulation-based verification [3]. Formal verification of hardware systems consists of property verification and equivalence checking. Equivalence checking uses formal methods to

determine whether given two designs are functionally equivalent or not, while property verification uses them to find proofs of validity for the provided properties of a design [4]. In property verification, the user first prepares formal properties of a design that express the design intent in formal logic. Model checkers then verify these properties. Model checkers exhaustively search for points in the Design Under Verification (DUV) state space that would fail the property currently being checked. The disadvantage of property verification using model checking is lengthy runtimes, especially when intricate RTL designs are involved.

Simulation-based verification relies on input vectors and simulation models of RTL designs for the verification . The idea behind the simulation-based approach is simulating RTL design using a set of input vectors, then comparing observed behaviour against the design specifications. Assertion-based verification is such approach that combines property checking with simulation-based verification. Instead of formally verifying the defined properties, the user implements them as assertions along with RTL design. Then, RTL design is simulated until a set of input vectors satisfying the defined assertions is found. Assertion-based verification like property checking aims to increase observability of the DUV, which corresponds to the ability to observe the internal state of the DUV contrary to the complete black-box methods that have limited observability [5]. This way, it is possible to reduce the time spent on debugging during verification.

Although it is possible to verify the functional correctness of a system with simulation-based verification, to prove that a design is error-free all input combinations should be defined. For any complex design, manually defining all input combinations is a long and error-prone process. Therefore, the idea of automated test input generation was suggested.

Automated input generation for hardware systems dates to the 1970s [6]. In automated input generation, generated inputs follow a predefined pattern. Following early works on automated test input generation, the researchers proposed input generation

methods using constraint solvers [7–10]. In constraint random input generation, the user defines a set of constraints that the input generator follows. Before each test is generated, the constraint solver is invoked to check whether the selected input satisfies all constraints on the given formal model of the RTL design. This approach still suffers from long runtimes to find proper constraints like model checking in formal methods.

Along with constraint random input generation, different studies further proposed another approach to improve input generation named directed input generation. In directed input generation, a selection algorithm is used by generators to create a bias towards inputs that allow the user to observe unexplored areas in an RTL design using the observations obtained from simulation. For selection algorithm works [11–15] propose genetic algorithm-based solutions. Genetic algorithms first change parts of input, model, or design randomly (mutate in other words), then use user-defined fitness function to decide whether mutations are improved observations or not. They provide generic solutions to explore interesting areas in the design. However, it is still possible for them unable to detect corner cases since it is hard for them to explore the entire RTL design.

For directed input generation, Fine and Ziv proposed a machine learning-based solution in [16]. In this work, the authors approached the topic as a machine learning problem and proposed a Bayesian decision tree method to improve input generation. Following their work, the application of different machine learning algorithms such as rule learning [17], neural networks [18], and reinforcement learning [19–22] become prominent.

In this thesis, we built a verification framework for assertion-based verification using reinforcement learning guided input generation. Our framework learns from individual assertions and finds specific test sequences satisfying given assertions. To the best of our knowledge, this is the first reinforcement learning-based generic framework for assertion-based verification.

1.2. Contributions

Our main contributions in this thesis are following:

(i) We developed a design agnostic assertion-based verification framework using a reinforcement learning guided input generation algorithm. To the best of our knowledge, this is the first generic framework in this field to use reinforcement learning.

(ii) We evaluated our framework with a 5-stage 64-bit RISC-V core and achieved significant performance improvement over random input generation.

1.3. Outline

The thesis outlined as follows: In Chapter 2, we provide background information on Linear Temporal Logic, Assertion-based Verification, and Reinforcement Learning. We discuss prior works related to our domain in Chapter 3. We explain our methodology and provide our results with an example in Chapter 4 and Chapter 5, respectively. Lastly, we conclude thesis with discussion on our overall contributions and future work in Chapter 6.

2. BACKGROUND

2.1. Linear Temporal Logic

Linear Temporal Logic (LTL) is a temporal logic that allows us to define propositional formulas referring to the future [23]. LTL models the time as sequences of states called computational paths continuing indefinitely into the future [24]. In our framework, the LTL syntax we use is

$$\phi := \top \mid \ \bot \ \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \mathrm{X}\phi \mid \mathrm{F}\phi \mid \phi \ \mathrm{U} \ \phi \tag{2.1}$$

where p is an atomic proposition and ϕ is a LTL formula.

2.1.1. Semantics of LTL

The systems verified by LTL are modelled as transition systems. The model of the DUV $\mathcal{M} = (S, \rightarrow, L)$ contains a set of states S, a transition relation $\rightarrow$ which is a binary relation on S, and a labelling function $L : S \rightarrow \mathcal{P}(\texttt{Atoms})$ where $\mathcal{P}(\texttt{Atoms})$ is power set of atomic propositions. Labelling function $L(s)$ gives us all atomic propositions that hold for the current state s.

A computational path π in model $\mathcal{M}$ is infinite sequence of state transitions. Satisfaction relation $\models$ between path $\pi = s_0 \rightarrow s_1 \rightarrow \ldots$ and LTL formula ϕ defined provided as follows:

- $\pi \models \top$
- $\pi \not\models \bot$
- $\pi \models p$ iff $p \in L(s_1)$
- $\pi \models \neg\phi$ iff $\pi \not\models \phi$
- $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$

- $\pi \models \mathrm{X}\phi$ iff $\pi^2 \models \phi$
- $\pi \models \mathrm{F}\phi$ iff there is some $i \geq 1$ such that $\pi^i \models \phi$

2.2. Assertion-based Verification

Assertion-based verification is a simulation-based verification methodology aiming to improve the observability and controllability of the RTL designs [25]. Controllability corresponds to the capability of activating specific parts in the design using only the design inputs. On the other hand, observability corresponds to observing the effects of inputs in the DUV. In simulation-based verification methodologies, RTL design itself is considered a black box entity. Therefore, although they have better controllability on the input ports of the design, they have lower controllability and limited observability on the internal structures. By integrating property-checking from formal methods with simulation-based verification, assertion-based verification intends to solve the low observability and controllability of the design internals.

2.2.1. Properties

Property in formal methods is the formal propositions derived from informal design specifications to define how specific design parts should work. In formal verification, the defined properties are verified using model checkers. Using the formal design model, model checkers exhaustively look for counterexamples that would fail the given properties. If there are not any counterexamples found, the properties are satisfied. Otherwise, counterexample means there is a design error, and it is possible to reproduce it following the points found in the counterexample. As model checking involves searching points in the internal structure of the design, it provides high observability for design internals. In the industry, properties are defined using Property Specification Language (PSL) [26]. PSL makes use of regular expressions to define a standardized syntax that allows users to express design specifications in the form of extended linear temporal logic.

2.2.2. Assertions

An assertion is an implementation of the property to validate design intent using an underlying verification tool. Like property-checking, assertions are used as targets for the verification process. Their definitions give insight into the internal components of the DUV. The difference between properties and assertions is that assertions are implementations of properties defined in the RTL implementation. As they are implemented in RTL design, it is possible to evaluate them using simulation-based methods. If a design error is activated during simulation, the related assertion will fail, and then engineers track the previous inputs and state transitions until that point to reproduce the error. This way, they can debug the RTL implementation without spending excessive time.

SystemVerilog [27] is a hardware design language used for implementing the high-level design. It is popular in the industry and supported by all commercial tools. Its specification comes with a particular set of syntax for defining assertions named SystemVerilog Assertion Language (SVA). SVA allows users to implement assertions using an extended form of linear temporal logic like PSL.

2.3. Reinforcement Learning

Reinforcement learning is a machine learning paradigm about mapping situations to actions using a reward signal [28]. Reinforcement learning aims to create a computational approach to learning from interactions when intelligent agents behave in an unknown environment. Unlike other machine learning paradigms, reinforcement learning does not need any labelled examples, nor is it concerned about finding hidden patterns in the collections of unlabelled data. Instead, it is concerned about maximizing the reward signal while reaching a goal and balancing exploration and exploitation. The general overview of the relation between environment and agent is presented in Figure 2.1.

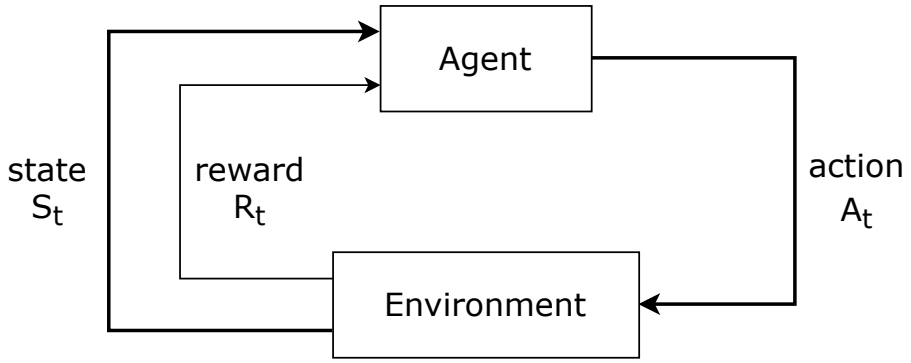

Figure 2.1. General interaction between agent and environment.

In reinforcement learning, an intelligent agent interacts with an unknown environment and obtains rewards from the environment to shape its policy. Reward signal defines the agent's goal and sends a numerical value called reward after each step. Reward value provides whether the agent's interaction was good or bad in the scope of the current interaction. If the interaction is good, the reward value becomes positive. Otherwise, it is negative. The agent aims to maximize total numerical reward in the long run. Reinforcement learning defines the interaction between agent and environment using Markov decision processes.

Rewards decide the desirability of the current environment states. In the long run, the desirability of states is determined by the value function. The value function uses the past and present information to obtain the value of the state while considering the future rewards.

The policy is the core of reinforcement learning. It is the decision-making part of the agent and determines how the agent should act at any given time. While it can be a simple function or lookup table, it may also involve complicated systems such as neural networks. For the current state s_t, policy decides on an action a_t by looking at the values of the following states which environment will be in when the agent interacts with the environment using the chosen action.

Some reinforcement learning systems have model that describe the environment. Models are probability distributions of state transitions that can predict next states and rewards. Systems that require planning might use the model-based methods, but there are also model-free methods that are not relying on any environment model and learns the model through trial-and-error.

2.3.1. Q-Learning

We use Q-Learning as the reinforcement learning algorithm. The main goal of Q-Learning is to maximize future expected rewards. It is a suitable algorithm for our framework as we may not evaluate the given assertion in the current simulation step. Therefore, we need to maximize future expected rewards. Although there are other reinforcement learning algorithms, such as those that utilize neural networks, they have a higher computation cost than Q-Learning.

Q-Learning uses a Q-Table to learn from the rewards given during runtime. Q-Table is the state-action table where each state-action pair has a Q-Value. We calculate the probability distribution of actions for the state s_t using Q-Values of actions for the state s_t. Then the policy samples an action a_t from this probability distribution. The Q-Values are calculated according to the quality function $Q : (SXA) \rightarrow \mathbb{R}$ which is

$$Q(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \max_a Q(s_{t+1}, a)) \tag{2.2}$$

where α and γ parameters are the algorithm's hyperparameters. We will explain these hyperparameters in Section 4.1.1 in detail.

3. RELATED WORKS

In recent years, the theory of applying reinforcement learning methodologies to test input generation has been explored in the testing domain, starting with software testing. Groce proposed a test input generation framework using Q-Learning to automatically generate test inputs for the unit test explicitly written for the software under test [29]. Similarly, Kim et al. propose in [30] a deep Q-networks-based test input generation for the software under test targeting improvement of the test coverage. Moreover, Reddy et al. propose a test input generation framework using Monte-Carlo Methods-based Q-Learning to apply reinforcement learning to property-based software testing [31]. The application of reinforcement learning was also extended to the GUI-testing. Koroglu and Sen proposed a Q-Learning-based automated GUI-testing framework in [32]. They have later proposed a framework for functional test generation of GUI applications using reinforcement learning. Their work proposed a novel policy algorithm and a reward shaping procedure, which we incorporated in our framework [33].

Alongside software testing, researchers also began to explore the application of reinforcement learning to hardware verification domain. Singh et al. propose a hardware verification framework that uses reinforcement learning for exploring unknown states of the DUV to generate the formal model to be used in model checkers later on [34]. Another work proposed by Wu et al. in [35] also applies reinforcement learning to find an optimal resource allocation strategy by exploring the DUV for FPGA design implementation using user-defined constraints. Although these frameworks incorporate reinforcement learning in the verification process, their use did not affect the test input generation process.

Bruns et al. propose a framework for early verification of RISC-V instruction set architecture (ISA) extensions using deep reinforcement learning to select valuable test programs that would trigger ISA flaws when executed on virtual prototypes im-

plemented according to extended ISA [19]. This framework was the first to use reinforcement learning to affect test generation, albeit only for selecting predefined test programs.

In recent works [20–22], reinforcement learning is used for test input generation. Pan et al. propose a framework for finding hardware trojans implanted in the RTL designs using reinforcement learning [20]. In this work, the authors focus on generating test inputs that would activate the logic gates, which would contain the injected trojans using a policy-gradient based reinforcement learning algorithm.

Pfeifer et al. proposed a reinforcement learning-based test input generation framework for shared memory verification [21]. Rather than learning for test inputs directly, the authors used deep reinforcement learning to differentiate which test groups generated using a constraint random input generator is more valuable to increase coverage of the given shared memory design. Their system accepts the test size with the maximum amount of shared location they can use as parameters. Then these parameters are updated according to the coverage observed during the predefined time limit. Although the authors observed higher coverage than the baseline generator they were using, their main aim was to learn which constraints are better for improving observed coverage rather than learning which test input to generate.

Choi et al. propose a deep reinforcement learning approach to generate test vectors capable of exploring all possible memory blocks in their DRAM designs [22]. The authors aim to fully verify DRAM designs using the deep reinforcement learning in this work. For this purpose, they prepared an action space using the input signal table and a state space consisting of the storage space contents. Furthermore, they implemented a Bayesian search optimization algorithm to reduce the size of generated test vectors. Although this work incorporates reinforcement learning to test input generation directly, the focus of this work was to generate efficient test vectors solely for the verification of the memory blocks present in their DRAM designs.

4. METHODOLOGY

In this chapter, we present our framework and the reinforcement learning-based input generation algorithm behind it. We provide the general structure of our framework and discuss how we modelled our verification process as a reinforcement learning process.

4.1. Reinforcement Learning Guided Input Generation

We mentioned that in simulation-based verification, input generation drives the design exploration in Chapter 2. Input pattern is an essential factor for finding design errors present in the hard-to-reach areas of the DUV because exploring those areas require particular patterns that contain different input orders with diverse input types. Randomly generated input sequences have a minuscule probability of fitting required patterns. Therefore, the problem of input generation focuses on selecting inputs to form the required patterns and explore the unknown areas of the DUV. Let us think of inputs as the actions taken by an agent and the entire testbench as an environment with which the agent is currently interacting. We can model the simulation-based verification as a reinforcement learning process visualized in Figure 4.1.

In the process visualized in Figure 4.1, the environment is the verification testbench that accepts an input and simulates the DUV. The agent interacts with this environment using the input stimuli generated according to design specifications. We mentioned previously that reinforcement learning is the problem of learning to accomplish the given goal using the reward signals obtained from the environment. Following this definition, we turned the input generation problem into a reinforcement learning problem by defining a reward signal R_t and providing a proper goal for the agent. In our framework, we designated the agent's goal as finding witnesses for the given assertions. We call the test sequence $t_s = a_0, a_1, \ldots, a_{t-1}, a_t$ a witness if the inputs satisfy the given assertion applied to the DUV in order.

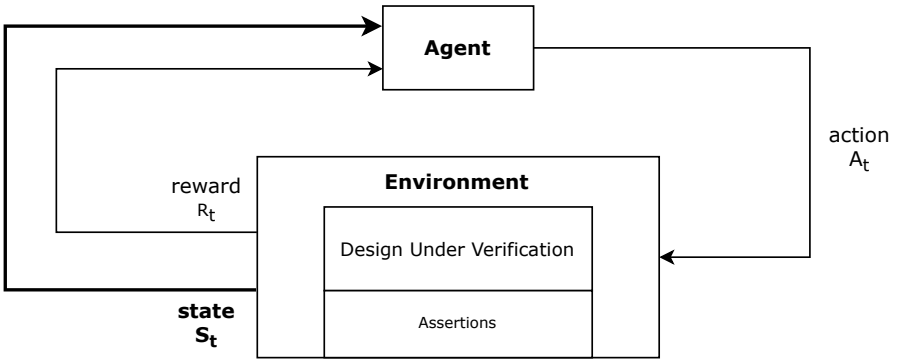

Figure 4.1. Model of the simulation-based verification as a reinforcement learning process.

Detailed overview of the input generation process is presented in Figure 4.2. In each episode, the agent's policy selects an action using the probability distribution calculated using the Q-Table. Then, the DUV is simulated using the selected action, and a numerical reward is calculated according to the assertion evaluation. If the test sequence satisfies the assertion, the reward becomes 1, and the simulation terminates. On the other hand, the reward becomes -1 if the assertion becomes unsatisfiable and the current episode terminates. If an assertion requires multiple inputs to be thoroughly evaluated, the immediate reward becomes zero or a partial value.

Each episode is an attempt to find a witness. They contain multiple steps. Each step is the equivalent of one simulation cycle. In our framework, one simulation cycle starts with the input generation, followed by the DUV simulation and assertion evaluation. Since our goal is to find a witness, the information we learned from each episode is preserved between episodes until we find the witness. The policy is the decision-maker of the agent. After reward calculation, an action-value is calculated using the obtained reward. This value is then used for Q-Table update. Q-Table is a lookup table for policy to infer probabilities of the subsequent actions for the current state. It contains the learned values calculated from the rewards obtained in recent episodes.

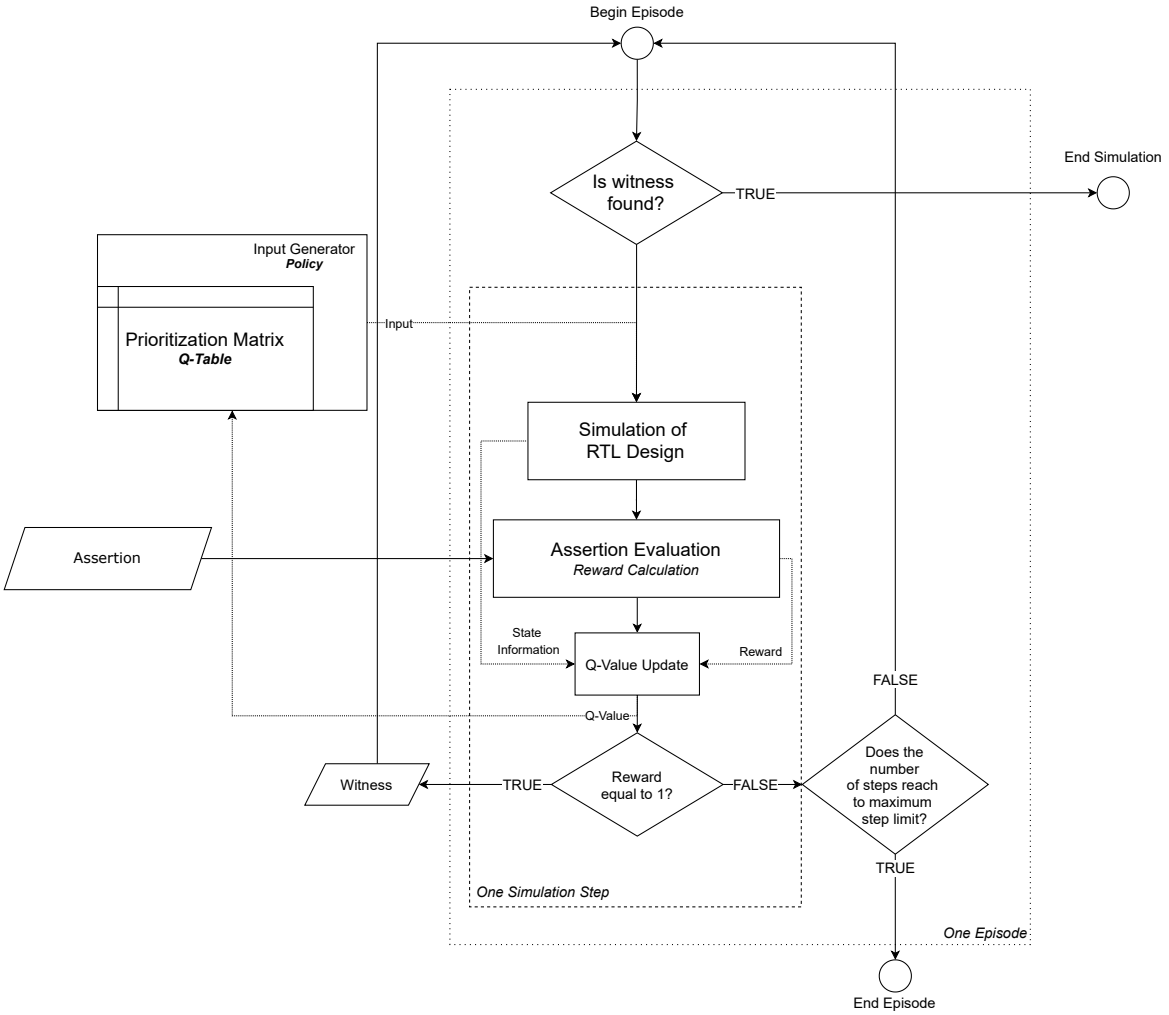

Figure 4.2. Detailed overview of the verification process of our framework.

Assertions are implementations of the design properties. In our framework, users define their assertions in the LTL form using the syntax presented in Section 2.1.

In our framework, we use Q-Learning as the basis of our input generation algorithm. Q-Learning is a model-free reinforcement learning algorithm. Q-Learning uses a quality function $Q : (S \times A) \rightarrow \mathbb{R}$ to calculate Q-Value, which is the action value indicating the desirability of the current action.

In reinforcement learning, the agent only obtains a reward if it reaches its goal. As we defined our goal to find a witness, our framework terminates right after the agent reaches its goal. Therefore, the agent only learns from the negative rewards.

Furthermore, in some cases, more episodes are required for the algorithm to find an optimal policy. To improve our input generation algorithm, we implement novel policy and reward calculation procedures proposed by Koroglu and Sen in [33].

We gave a detailed overview of how our framework works. In the following sections, we will explain the algorithm we implemented in detail.

4.1.1. Implementation

Our framework integrates with RTL design implementation by only providing a thin wrapper to define input space, state variables, and input application method. We provide the pseudo-code of the primary procedure for our framework in Algorithm 4.1.1. Our framework has five requirements. These requirements are the list of assertions, the maximum number of episodes E, the maximum number of steps T, the instance of the DUV Ω, and the input specification θ. The algorithm runs for each assertion. Since we need assertions to calculate reward, the user should provide the list of assertions before the algorithm begin working.

The maximum number of episodes E corresponds to the maximum number of attempts the framework makes before it terminates. If the maximum number of episodes is too small, the framework's attempts might come up dry and terminate without any witnesses. Furthermore, the framework might spend unnecessary time on unsatisfiable assertions if the maximum number of episodes is set too high. Due to these situations, it is crucial how the episode limit is defined.

The maximum number of steps T corresponds to the maximum number of simulation steps taken by the algorithm during one episode. In our framework, the step means the complete simulation cycle, which is precisely the passing of one clock cycle. The assertions using the next operator rely on future states to be evaluated. Therefore, the framework cannot find a witness if we set the maximum number of steps too small. Similarly, if we choose the T too high, the framework will lose too much time on a

single episode, and the resulting witness will be unnecessarily long.

To integrate an RTL design with our framework, the instance of the DUV Ω and the input specifications θ are required. The instance of the DUV is the programmatic and clock accurate model of the DUV provided by the underlying simulation tool. We use this model to expose the internal wires and structures of the DUV to our framework. Furthermore, we obtain fine-grained control over simulation steps using the programmatic model. We need the input specifications for our policy. The policy is the decision-making part of the framework. Using the Q-Table, the policy infers probabilities of the next possible actions for the current state and selects one of them. Policy implements the action space and the actions using the input specification provided. The resulting test sequences contain actions only defined in the input specifications.

Before starting the execution, we initialize our witness list in Line 1 of the Algorithm 4.1.1. We search for witnesses for the given assertions iteratively. Before starting the first episode, we initialize the Q-Table and policy with the input specifications and the Q-Table in Lines 4 and 5, respectively. In the Q-Table, the rows represent the state of the DUV, and the columns represent the actions. Action space comes from the input specifications. Therefore, we can allocate space for them in the Q-Table. But defining state space for intricate RTL designs is almost impossible. The good thing with reinforcement learning is that we do not need to define state-space explicitly. Model-free reinforcement algorithms can explore unknown states during runtime. Since we are using Q-Learning, we explore the state space during runtime by leveraging this property. Initial action values for each discovered state will be zero in the Q-Table. In the beginning, Q-Table will be empty because the policy currently knows nothing. As more episodes progress, the agent will obtain more rewards, and the policy will be more knowledgeable about selecting better actions towards maximizing the expected reward.

Require: List of assertions = $\{\phi_1, \phi_2, \ldots, \phi_n\}$
Require: Maximum number of episodes E
Require: Maximum number of simulation steps T
Require: Instance of DUV Ω
Require: Input Specification θ
Ensure: Witnesses = $ts_1, ts_2, \ldots, ts_n$

witnesses = $\emptyset$
for all assertion ϕ_n **in** list of assertions **do**
    EpisodeIndex $\leftarrow 0$
    $Q \leftarrow$ EmptyTable()
    $\pi \leftarrow$ InitSoftmaxPolicy(θ, Q)
    while EpisodeIndex $< E$ **and** witness not found **do**
        $ts_n = \emptyset$
        SimulationStep $\leftarrow 0$
        ResetDevice(Ω)
        while SimulationStep $< T$ **do**
            $s_t \leftarrow$ GetState(Ω)
            $a_t \leftarrow$ SelectAction(π, s_t)
            $ts_n \leftarrow ts_n \bigcup a_t$
            $L(s_t) \leftarrow$ Evaluate(Ω, a_t)
            $s_{t+1} \leftarrow$ GetState(Ω)
            $r_t \leftarrow$ CalculateReward(ϕ, $L(s_t)$)
            $Q(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \max_a Q(s_{t+1}, a))$
            UpdateQTable(π, $Q(s_t, a_t)$)
            if $r_t = 1$ **then**
                witnesses = witnesses $\bigcup ts_n$
                end episode with witness ts_n
            else if $r_t = -1$ **then**
                end episode

Figure 4.3. The main algorithm of RL-Guided input generation.

We begin each episode by resetting the test sequence and the DUV to its initial state as stated in Line 7 and Line 9, respectively. Starting from the initial state allows us to trace the simulation steps and state transitions after each input in the test sequence. Traceability of the state transitions is a necessity. Because, as we mentioned before, one of the main advantages of assertion-based verification is improving the observability of the DUV's internal states. With the found test sequence, the user shall observe all the state transitions leading to the design error. If we were to start from a random state, the user would not retrace the state transitions because the initial state would be unknown. Similarly, we need to reset the test sequence to an empty set because of the same reason.

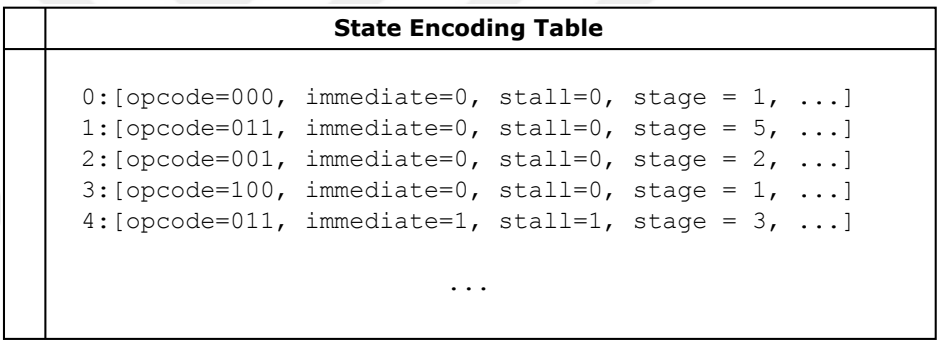

Figure 4.4. Example state encoding table. Here, state vector $s = [\texttt{opcode} = 000, \texttt{immediate} = 0, \texttt{stall} = 0, \texttt{stage} = 1, \dots]$ will be compared against the elements of the table. If it is previously appended to the table, its table index will return. In this example, s exists in the table. Therefore, its index will be 0.

Our framework starts the simulation steps starting from the reset state. In each step, we first obtain the current state from the programmatic model of the DUV in Line 11. The wrapper around the programmatic model provides the GetState procedure which returns a list of values corresponding to the internal wires. The user implements the GetState procedure in the wrapper. The framework then encodes the states provided by this procedure and obtains state index. For example, let us assume we have a state vector $s = [\texttt{opcode} = 000, \texttt{immediate} = 0, \texttt{stall} = 0, \texttt{stage} = 1, \dots]$

at time t. GetState procedure returns the vector s, and then the framework would encode this vector by comparing it with the values in the state encoding table. If the vector exists in the table, then the encoder would return the position of the vector in the table. Otherwise, the encoder would append the vector to the table and return the position of the vector. An example state encoding table provided in Figure 4.4.

The policy decides the next action using the state index in Line 12. Since the policy is initialized with the input specifications, it knows the action space of the DUV. We add the selected action to the test sequence and simulate the DUV using the selected action. The policy is inferring a probability distribution for the next possible actions using the action values of the current state s_t from the Q-Table. The policy then samples an action from the distribution using the probability distribution inferred.

Evaluate procedure simulates the DUV for a full clock cycle. It returns the list of atomic propositions that hold the system derived from the internal wires and registers of the DUV. The programmatic model provides Evaluate procedure. This procedure propagates the changes made to the inputs to the DUV. Both rising edge and falling edge should be applied to the clock signal to simulate a clock cycle completely. The programmatic model requires the user to handle edge changes. The wrapper around the Evaluate procedure controls the clock signal and allow the simulation to continue for a full clock cycle. We mentioned that atomic propositions are the smallest part of the proposition that may hold the system. The Evaluate procedure returns the list of propositions that are holding the system called the labels $L(s_t)$. The labels only provide us with the atomic propositions that hold the system. Therefore, we can evaluate the assertion ϕ_n by substituting the atomic propositions that belong to the $L(s_t)$ with true and those that do not belong with false. If the next operator encapsulates the part of the atomic propositions, then we cannot substitute them as they cannot be evaluated at the current time. We use the labels $L(s_t)$ and the assertion ϕ_n for calculating immediate reward.

In reinforcement learning, the agents always try to obtain positive rewards, and it obtains positive rewards when it accomplishes its goal. In our design, simulation terminates when we reach our goal because it is unnecessary to continue after finding a witness. Since we terminated the simulation after reaching our goal, our policy can only learn from the negative rewards. This situation slows down the learning process, so to prevent this, we decided to grant partial rewards using the labels obtained from Evaluation procedure. CalculateReward procedure in Line 16 calculates the partial reward using the current assertion ϕ_n, and the labels $L(s_t)$. The calculated reward is $r_t \in [-1, 1]$.

We only give reward $r_t = 1$ when the current test sequence ts_n satisfies the current assertion ϕ_n. Therefore, after the Q-Value update, we check for reward value. If the reward is 1, then we terminate the simulation with the test sequence ts_n. If the reward is -1, it means that we have reached a state where it is impossible to satisfy the current assertion ϕ_n. So, we terminate our current episode and start a new one, preserving our knowledge.

In our framework, the goal is finding witnesses for given assertions, and after we find a witness, our framework terminates. Although we can still learn using only the negative rewards, we would lose the information we gather via intermediate observations. Intermediate observations occur when the selected action does not lead to a witness. This situation happens especially when the assertion ϕ_n contains the next operator, since we must evaluate them in the consecutive states. Therefore, we need a way to direct policy towards selecting actions that are more likely to be part of the witnesses. For this purpose, we use specification-based reward shaping procedure and ϵ-greedy softmax policy proposed by Koroglu and Sen in [33].

The atomic propositions encased with the next operator is evaluated in the following rather than the current step while the rest is evaluated in the current step. Typically, the agent would obtain zero rewards in this situation. As a result, the policy would not learn anything despite valuable observations. Using the reward shaping al-

gorithm introduced in [33], we can derive partial rewards for our valuable intermediate observations using reward shaping. This way, we can make the learning process more efficient.

After obtaining a numerical reward value, we use this reward to calculate the action value and update the Q-Table accordingly, as presented in Line 17 and Line 18, respectively. In Q-Learning, we use a quality function $Q : (S \times A) \rightarrow \mathbb{R}$ to calculate action value $Q(s_t, a_t)$ at the time t. We use this value which is also called Q-Value, for the learning process. We calculate the Q-Value as

$$Q(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \max_a Q(s_{t+1}, a)) \tag{4.1}$$

where the learning rate α and the discount factor γ are the hyperparameters of the Q-Function. Learning rate allows us to determine how much our algorithm should learn from the newly acquired information. If the learning rate is set too high, the algorithm would only learn from the recent information and ignore the prior knowledge. Exploring other possibilities during the runtime rely on prior knowledge. Therefore, a high learning rate would cause the policy to disregard other possibilities and select an action using only the most recent observation. We use $\alpha = 0.1$ as our learning rate in our algorithm.

The discount factor γ allows us to define the importance of future rewards. The part denoted by $\max_a Q(s_{t+1}, a_t)$ is the maximum expected future reward. If we want to increase the importance of immediate rewards, we set γ small. The high discount factor allows the algorithm to aim for the maximum reward in the long run. In our framework, long test sequences may be required to satisfy the given assertions along with short sequences. Therefore, selecting this value carefully to create a proper balance between learning from the expected future and immediate rewards is important. We set $\gamma = 0.9$ in our implementation.

5. EXPERIMENTAL RESULTS

5.1. Experimental Setup

5.1.1. Overview

We designed our framework to work with any RTL design. We separated all the necessary components into modules and prepared a generic wrapper. We use a wrapper to interface the programmatic model with the framework's verification environment. We present a general overview of our module hierarchy and how they interact with each other in Figure 5.1.

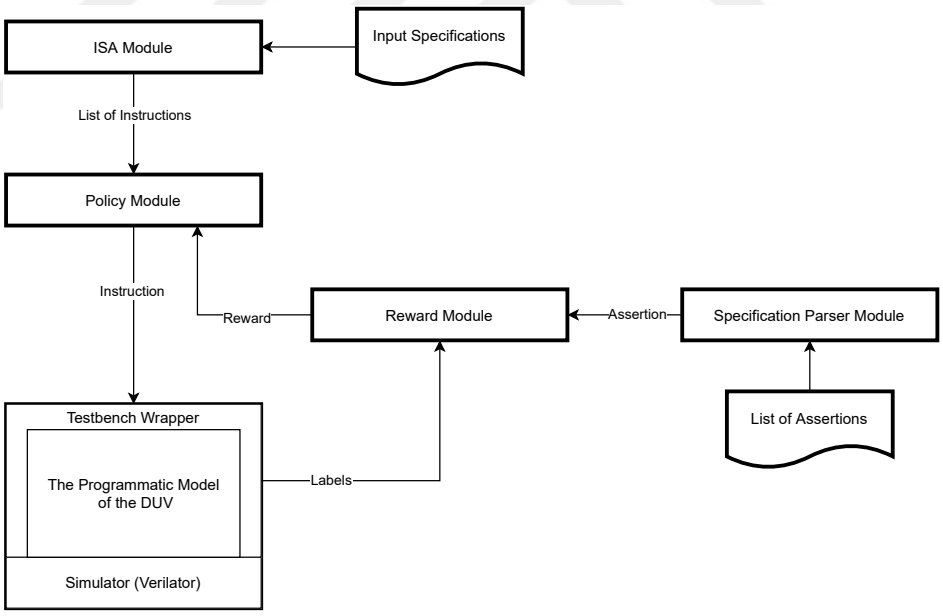

Figure 5.1. A general overview of module hierarchy.

We prepared our test environment using an in-order 64-bit 5-stage RISC-V core [36]. The policy module is our input generator. It selects an input from the list of possible inputs, and the instruction set architecture (ISA) module provides the input chosen according to the input specifications. ISA module is the implementation of

the given input specifications. Testbench wrapper accepts the input generated by the policy module and applies it to the programmatic model of the DUV. Afterwards, the simulator simulates the model using the applied input then generates labels given to the reward module. The reward module accepts labels and gets an assertion from the list of assertions. Specification parser module parses the assertion next-in-line into abstract syntax tree then gives the resulting abstract syntax tree to the reward module. The reward module then runs the projection procedure and calculates the immediate reward using the labels and the given assertion. It provides the calculated reward to the policy module, updating its Q-Table using the reward.

As a simulator, we use an open-source simulation tool called Verilator [37]. Verilator compiles the RTL design into time accurate C++ model of the given design. This process is called *verilation*. The user then simulates the *verilated* model using the exposed APIs. The *verilation* process gives us fine-grained control over simulation steps which is essential for our framework. On top of that, the *verilated* model is time accurate, so the traced simulation steps are reproducible with any other simulator. In our framework, we automatically *verilate* the RTL design and instantiate it in the wrapper we use for interfacing the programmatic model with the other modules.

5.1.2. Input Specifications

We used a custom 64-bit 5-stage RISC-V core for the experiments. RISC-V is an open-source RISC-based instruction set architecture. We used RISC-V RV32IM instruction set architecture specifications [38]. We chose RV32IM because 32-bit and 64-bit RISC-V architectures use the same base ISA with the only difference being the register width. The specification indicates a few small extensions in some instructions, such as multiplication and division. We provide the RV32IM instruction set architecture in Table 5.1.

The RISC-V RV32IM ISA specifies six types of instructions with 48 instructions. The instructions are grouped according to their parameter types and functionality. The

ISA also defines 32 registers, each indexed from 0 to 31. Some registers such as `x0` are special and cannot store any data.

Table 5.1. Overview of RV32IM instruction set architecture.

Instruction Type	**Bit Fields**						
Register	`funct7`		`rs2`	`rs1`	`funct3`	`rd`	`opcode`
Immediate	`imm[11:0]`			`rs1`	`funct3`	`rd`	`opcode`
Upper Immediate	`imm[31:12]`					`rd`	`opcode`
Store	`imm[11:0]`			`rs1`	`funct3`	`imm[4:0]`	`opcode`
Branch	`imm[12]`	`imm[10:5]`	`rs2`	`rs1`	`funct3`	`imm[4:1]` `imm[11]`	`opcode`
Jump	`imm[10]`	`imm[10:1]`	`imm[11]`	`imm[19:12]`		`imm[4:1]` `imm[11]`	`opcode`

5.1.3. State Information

We need to create distinct states for our algorithm to work successfully. We used the internal information of the DUV to define a state vector. This state vector contains information such as instruction type, current stage of the processor, whether data is being written to the memory or not, and stall information. Combining this information gives us a distinctive state that we encode and use for the learning process. An example state vector along with an example state encoding table is presented in Figure 4.4.

5.1.4. Example

We now explain how our framework works in the test environment using a small example from the official RISC-V specifications. We found a design bug related to the jump operation that occurs in our DUV. According to RISC-V specifications, jump instruction gets an address from either a register or using an immediate. Before the jump operation, instruction stores the return address to the destination register. If the jump instruction has a misaligned or invalid address, the jump operation should trigger a trap, and the destination register should not be modified.

Specification In case of jump address is misaligned, and trap occurred destination register `rd` should not be modified.

Assertion We formalize this specification as a design bug, expecting `rd` to be modified after the operation.

LOAD_IMMEDIATE $\wedge \mathbf{X}(\text{JUMP} \wedge \mathbf{X}(\mathbf{F}(\text{RD} \neq \text{PREV_RD})))$

- LOAD_IMMEDIATE is an atomic proposition that only becomes true when applied instruction is one of the immediate load operations such as `LW`, `LH` or `LB`.
- JUMP is an atomic proposition that only becomes true when applied instruction is a jump operation.
- RD is value of the return register that applied jump operation uses, and PREV_RD is the previous value of the same register before the jump operation.

We first need to load a non-existent address from memory to register. Afterwards, we need to jump to that address. A trap will occur during the jump operation, and we observe that the destination register is eventually modified. According to the RISC-V specifications, this is a side effect of the operation, and it is a design bug.

Our DUV has a predefined 64 KB memory with 1024 words and 64-bit word size. Addresses larger than 1024 would be invalid. Our framework found a witness after 260 episodes on average for this assertion. The execution steps are presented in Table 5.1.4. For the sake of simplicity, we represent the atomic propositions as follows: $p = \text{LOAD_IMMEDIATE}, q = \text{JUMP}, r = (\text{RD} \neq \text{PREV_RD})$. The atomic propositions will be evaluated at the seventh step at the earliest as jump operation will stall to the pipeline. In the first episode, we selected `DIVU` which was different from what we were expecting. Therefore, the assertion is evaluated as false($\bot$) and episode is terminated. While calculating reward, the assertion is expanded using the equivalence of $\mathbf{F}(\phi) = \top \mathbf{U} \phi$ and expansion law. Resulting assertion is

$$\phi = p \wedge \mathbf{X}(q \wedge \mathbf{X}(\neg(\neg r \wedge \neg(\top \wedge \mathbf{X}(\top \wedge r))))). \tag{5.1}$$

Table 5.2. Execution steps for the example run.

	Episode 1		$\phi = p \wedge \mathbf{X}(q \wedge \mathbf{X}(\mathbf{F}(r)))$	
t = 0	$a_0 =$ DIVU	$L = \{\}$	$\phi = \perp$	$r = -1$
...				
	Episode 47		$\phi = p \wedge \mathbf{X}(q \wedge \mathbf{X}(\mathbf{F}(r)))$	
t = 0	$a_0 =$ LB	$L = \{p\}$	$\phi = q \wedge \mathbf{X}(\mathbf{F}(r))$	$r = 0.2$
t = 1	$a_1 =$ SLTI	$L = \{\}$	$\phi = q \wedge \mathbf{X}(\mathbf{F}(r))$	$r = -1$
...				
	Episode 260		$\phi = p \wedge \mathbf{X}(q \wedge \mathbf{X}((r)))$	
t = 0	$a_0 =$ LB	$L = \{p\}$	$\phi = q \wedge \mathbf{X}(\mathbf{F}(r))$	$r = 0.2$
t = 1	$a_1 =$ JALR	$L = \{q\}$	$\phi = q \wedge \mathbf{X}(\mathbf{F}(r))$	$r = 0.33$
t = 2	$a_2 =$ REMU	$L = \{\}$	$\phi = \mathbf{F}(r)$	$r = 0$
...				
t = 6	$a_6 =$ SRA	$L = \{r\}$	$\phi = \top$	$r = 1$

After a step taken, the evaluation result was false($\perp$). Therefore, the reward was -1 and the episode is terminated. In later episodes, we encountered an immediate load operation LH and obtained 0.2 as the reward. We calculate the reward as

$$\begin{aligned} r_t &= \frac{\text{ABS}(|\phi| - |\phi_{projected}|)}{|\phi| + |\phi_{projected}|} \\ r_t &= \frac{\text{ABS}(3-2)}{|3+2|} \\ r_t &= 0.2. \end{aligned} \tag{5.2}$$

The original assertion had 3 atomic propositions. When $a_0 =$ LH selected, the projected assertion becomes $\phi_{projected} = \text{JUMP} \wedge \mathbf{X}(\mathbf{F}(\text{RD} \neq \text{PREV_RD}))$. Therefore, it has 2 atomic propositions. The reward becomes 0.2 in this step calculated using the distance metric we mentioned.

The resulting test sequence is $ts = \{a_0 = \texttt{LB}, a_1 = \texttt{JALR}, a_2 = \texttt{REMU}, a_3 = \texttt{SLTU}, a_4 = \texttt{XOR}, a_5 = \texttt{SLTU}, a_6 = \texttt{SRA}\}$

5.1.5. Experimental Results

In this section, we present the effectiveness of our framework. We prepared four different assertions targeting different specifications for testing purposes. We provide our test cases in Table 5.3.

Table 5.3. List of assertions used for evaluation.

	Assertions
ϕ_1	LOAD_IMMEDIATE $\wedge$ $\mathbf{X}$(STORE) $\wedge$ $\mathbf{X}$($\mathbf{X}$(STALL))
ϕ_2	BRANCH $\wedge$ $\mathbf{X}$($\mathbf{X}$(STALL))
ϕ_3	STORE $\wedge$ $\mathbf{X}$(STORE) $\wedge$ $\mathbf{X}$($\mathbf{X}$($\neg$STALL))
ϕ_4	LOAD_IMMEDIATE $\wedge$ $\mathbf{X}$(JUMP $\wedge$ $\mathbf{X}$($\mathbf{F}$(RD $\neq$ PREV_RD)))

Descriptions for the presented assertions are:

- ϕ_1 checks the pipeline stalling mechanism for the write-after-read hazards working properly. LOAD_IMMEDIATE proposition only becomes true when an immediate load instruction is applied to the DUV. Similarly, STORE proposition only becomes true when store type instruction is applied to the DUV. STALL proposition becomes true when pipeline is stalled. The pipeline is stalled when the source register of the store operation is the same as the destination register of the load operation. Since we do not know the source and destination registers until the decode stage of the pipeline, the STALL proposition becomes true after the third clock cycle.
- ϕ_2 checks the pipeline stalling mechanism for control hazards working properly. BRANCH proposition only becomes true when branch type instruction is applied. STALL proposition is the same one from ϕ_1. Our DUV solves control hazards

by pipeline bubbling. Therefore, pipeline should be stalled after branch operation. The STALL proposition becomes true after the third clock cycle, as the instruction type is obtained in decode stage.

- ϕ_3 checks for write-after-write hazards. It checks whether there are any stalls after two consecutive store type instructions. Pipeline should not be stalled, since the pipeline has writeback stage according to the specifications.
- ϕ_4 is the same assertion we used in the Section 5.1.4. It checks for the side effect that occurs after the failed jump operation.

We run our algorithm and compare the results against random input generation. We set the maximum number of steps as $T = 10$ and the maximum number of episodes as $E = 10000$. We run the experiments ten times for random input generation and our framework, respectively.

Table 5.4. Average Episodes Across 10 Runs.

	ϕ_1	ϕ_2	ϕ_3	ϕ_4
Random	4147.1	36.3	299.5	1231.9
RL	452.7	26.6	96.6	252.9

Table 5.5. Average Times Across 10 Runs.

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	**Total**
Random	9.32s	0.09s	0.66s	2.80s	**12.87s**
RL	1.55s	0.07s	0.26s	0.82s	**2.7s**

We present average episodes required and average time passed to find the witnesses in Table 5.4 and Table 5.5, respectively. From Table 5.4 we observed that our framework required fewer attempts to find witnesses for all assertions on average. Similarly, we observed that our framework was also 5 times faster than the random input generation on average. The resulting test sequences are presented in Table 5.6.

Table 5.6. Resulting test sequences for the assertions provided in Table 5.3

	Test Sequences
ts_1	$\{a_0$ =LH, a_1 =SW, a_2 =LB, a_3 =DIVU$\}$
ts_2	$\{a_0$ =BLTU, a_1 =ORI, a_2 =SLTI$\}$
ts_3	$\{a_0$ =SB, a_1 =SW, a_2 =REM$\}$
ts_4	$\{a_0$ =LB, a_1 =JALR, a_2 =REMU, a_3 =SLTU, a_4 =XOR, a_5 =SLTU, a_6 =SRA$\}$

5.1.5.1. Impact of the maximum episode limit. To demonstrate the impact of the maximum episode limit, we set the $E = 300$ and run the framework for the same assertions provided in Table 5.3. The results are provided in Table 5.7.

Table 5.7. Average Episodes for Our Framework Across 10 Runs.

ϕ_1	ϕ_2	ϕ_3	ϕ_4
300.0	30.9	133.1	258.0

Table 5.8. Number of Failures Across 10 Runs.

ϕ_1	ϕ_2	ϕ_3	ϕ_4
10	0	0	4

.

When we looked at the results presented in Table 5.7, we saw that our average episode count did not change much. Still, from the Table 5.8 we observed that our framework was unable to find witnesses for assertions ϕ_1 and ϕ_4 in some runs even though it was able to find suitable witnesses for each runs previously. As we mentioned before, this situation occurred because the framework hits the limit before finding a witness when we set the episode limit too small. Therefore, the framework was not able to fully utilize its prior knowledge. If it were to run a few more episodes, it would be able to use its prior knowledge to find suitable witnesses for each assertion, as we observed in Table 5.6.

5.1.5.2. Impact of the assertion complexity. We tested the impact of the assertion complexity by defining a new set of assertions that heavily use temporal operators. When assertions use temporal operators more, the attempts and time required for also affected. We used assertions provided in Table 5.9 to test the impact of the assertion complexity. Although these assertions can be satisfied in a few clock cycles, it is harder to find witnesses for them due to the extensive usage of temporal operators. In this table, R_TYPE_ADD atomic proposition becomes true when an register-type add operation occurs.U_TYPE proposition becomes true when any upper-immediate type instruction is selected. The rest of the propositions are as same as before.

Table 5.9. List of assertions used to test the impact of the assertion complexity.

	Assertions
ϕ_5	R_TYPE_ADD $\wedge$ **X**(STORE **U** (LOAD_IMMEDIATE $\wedge$ **X**(BRANCH)))
ϕ_6	R_TYPE_ADD $\wedge$ **X**(STORE **U** ((LOAD_IMMEDIATE $\wedge$ **X**(BRANCH)) $\wedge$ **X**(STORE **U** (LOAD_IMMEDIATE $\wedge$ **X**(BRANCH)))))
ϕ_7	R_TYPE_ADD $\wedge$ **X**((STORE **U** LOAD_IMMEDIATE) $\wedge$**X**((STORE **U** LOAD_IMMEDIATE) $\wedge$ **X**(R_TYPE_ADD **U** BRANCH)))
ϕ_8	(R_TYPE_ADD $\wedge$ **X**(STORE)) **U** (R_TYPE_ADD $\wedge$ **X**(BRANCH $\wedge$ **X**(U_TYPE)))

Table 5.10. Resulting test sequences for the assertions provided in Table 5.9

	Test Sequences
ts_5	$\{a_0$ =ADD, a_1 =LB, a_2 =BLTU$\}$
ts_6	$\{a_0$ =ADD, a_1 =SB, a_2 =SH, a_3 =LB, a_4 =BEQ$\}$
ts_7	$\{a_0$ =ADD, a_1 =SB, a_2 =LW, a_3 =BGEU$\}$
ts_8	$\{a_0$ =ADD, a_1 =BLT, a_2 =LUI$\}$

The average episodes and runtimes for each assertion are presented in Table 5.11 and Table 5.12, respectively. We set the maximum number of steps as $T = 10$ and the

maximum number of episodes as $E = 10000$. Similarly, we run the experiments ten times for random input generation and our framework, respectively. The resulting test sequences are provided in Table 5.10.

Table 5.11. Average Episodes Across 10 Runs.

	ϕ_5	ϕ_6	ϕ_7	ϕ_8
Random	3516.6	10000.0	7385.6	3607.4
RL	302.8	921.4	519.4	303.7

Table 5.12. Average Times Across 10 Runs.

	ϕ_5	ϕ_6	ϕ_7	ϕ_8	**Total**
Random	7.53s	24.42s	18.65s	10.14s	**60.74s**
RL	0.90s	3.65s	1.94s	1.12s	**7.61s**

We saw from Table 5.11 that the complex assertions require more attempts to find suitable witnesses, even though the resulting test sequences were not that long. The main reason for this is the extensive usage of temporal operators. This impact was also observed in runtimes as presented in Table 5.12. Compared to the random test generation, our framework achieved significantly better performance. Moreover, we obtained from Table 5.13 that the random input generation was unable to find a witness for some assertions in the given episode limits, whereas our framework managed to find suitable witnesses at each runs.

Table 5.13. Number of Failures Across 10 Runs.

	ϕ_5	ϕ_6	ϕ_7	ϕ_8
Random	0	10	6	1
RL	0	0	0	0

6. CONCLUSION

Functional verification is a crucial part of the digital design process. Debugging and verifying the design specifications takes a considerable amount of time in the verification process, especially with complicated designs. Assertion-based verification effectively reduces time spent on debugging during verification by implementing design specifications as assertions and finding input sequences that satisfy those assertions during the simulation. Design errors are then traced using these assertions allowing faster debugging.

In this thesis, we present a verification framework that uses a reinforcement learning-based input generation algorithm to improve verification speed. We use Q-Learning to learn from the given assertions to generate inputs. Furthermore, we implement ϵ-greedy softmax policy and LTL-based reward shaping procedures first proposed by Koroglu and Sen in [33] to improve learning speed. We evaluated our framework using a 64-bit 5-stage RISC-V core and achieved significant performance improvement over random input generation.

In the future, we will apply our framework to more complicated designs with different reinforcement learning approaches such as multi-agent reinforcement learning or deep reinforcement learning. Moreover, we will allow our framework to find witnesses for each assertion in parallel instead of sequentially to improve overall performance further.

REFERENCES

1. Mano, M. M. and M. Ciletti, *Digital Design: With An Introduction to The Verilog HDL*, Pearson, 2013.

2. Foster, H., "Part 8: The 2020 Wilson Research Group Functional Verification Study", `https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verification-study/`, accessed in November 2021.

3. Kropf, T., *Introduction to Formal Hardware Verification*, Springer Science & Business Media, Oct. 1999.

4. Lam, W. K. C., *Hardware Design Verification: Simulation and Formal Method-based Approaches*, Prentice Hall Modern Semiconductor Design Series, Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, 2005.

5. Foster, H. D., A. C. Krolnik and D. J. Lacey, *Assertion-Based Design*, Springer Science & Business Media, May 2004.

6. Duhamel, P. and J. Rault, "Automatic Test Generation Techniques for Analog Circuits and Systems: A Review", *IEEE Transactions on Circuits and Systems*, Vol. 26, No. 7, pp. 411–440, Jul. 1979.

7. Shimizu, K. and D. Dill, "Deriving A Simulation Input Generator and A Coverage Metric from A Formal Specification", *Proceedings of the 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*, pp. 801–806, Jun. 2002.

8. Adir, A., E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov and A. Ziv, "Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification", *IEEE Design Test of Computers*, Vol. 21, No. 2, pp. 84–93, Mar. 2004.

9. Kitchen, N. and A. Kuehlmann, "Stimulus Generation for Constrained Random Simulation", *2007 IEEE/ACM International Conference on Computer-Aided Design*, pp. 258–265, Nov. 2007.

10. Haedicke, F., H. M. Le, D. Große and R. Drechsler, "CRAVE: An Advanced Constrained Random Verification Environment for SystemC", *2012 International Symposium on System on Chip (SoC)*, pp. 1–7, Oct. 2012.

11. Faye, P. and E. Cerny, "Improved Design Verification by Random Simulation Guided by Genetic Algorithms", P. J. Ashenden, J. P. Mermet and R. Seepold (Editors), *System-on-Chip Methodologies & Design Languages*, pp. 81–95, Springer US, Boston, MA, 2001.

12. Chupilko, M., A. Kamkin, A. Kotsynyak and A. Tatarnikov, "MicroTESK: Specification-Based Tool for Constructing Test Program Generators", O. Strichman and R. Tzoref-Brill (Editors), *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science, pp. 217–220, Springer International Publishing, Cham, 2017.

13. Laeufer, K., J. Koenig, D. Kim, J. Bachrach and K. Sen, "RFUZZ: Coverage-directed Fuzz Testing of RTL on FPGAs", *Proceedings of the International Conference on Computer-Aided Design*, pp. 1–8, ACM, San Diego California, Nov. 2018.

14. Canakci, S., L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele and A. Joshi, "DirectFuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing", *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 529–534, Dec. 2021.

15. Klemmer, L. and D. Große, "EPEX: Processor Verification by Equivalent Program Execution", *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, pp. 33–38, ACM, Virtual Event USA, Jun. 2021.

16. Fine, S. and A. Ziv, "Coverage Directed Test Generation for Functional Verification using Bayesian Networks", *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pp. 286–291, Association for Computing Machinery, New York, NY, USA, Jun. 2003.

17. Chen, W., L.-C. Wang, J. Bhadra and M. Abadir, "Simulation Knowledge Extraction and Reuse in Constrained Random Processor Verification", *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, May 2013.

18. Wang, F., H. Zhu, P. Popli, Y. Xiao, P. Bodgan and S. Nazarian, "Accelerating Coverage Directed Test Generation for Functional Verification: A Neural Network-based Framework", *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pp. 207–212, ACM, Chicago IL USA, May 2018.

19. Bruns, N., D. Große and R. Drechsler, "Early Verification of ISA Extension Specifications using Deep Reinforcement Learning", *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, pp. 297–302, ACM, Virtual Event China, Sep. 2020.

20. Pan, Z., J. Sheldon and P. Mishra, "Test Generation using Reinforcement Learning for Delay-based Side-Channel Analysis", *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–7, Nov. 2020.

21. Pfeifer, N., B. V. Zimpel, G. A. G. Andrade and L. C. V. dos Santos, "A Reinforcement Learning Approach to Directed Test Generation for Shared Memory Verification", *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 538–543, Mar. 2020.

22. Choi, H., I. Huh, S. Kim, J. Ko, C. Jeong, H. Son, K. Kwon, J. Chai, Y. Park, J. Jeong, D. S. Kim and J. Y. Choi, "Application of Deep Reinforcement Learning to Dynamic Verification of DRAM Designs", *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 523–528, Dec. 2021.

23. Pnueli, A., "The Temporal Logic of Programs", *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science, FOCS*, Vol. 1977-, pp. 46–57, 1977.

24. Huth, M. and M. Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems*, Book, Whole, Cambridge University Press, Cambridge, 2nd edn., 2004.

25. Foster, H., "Applied Assertion-based Verification: An Industry Perspective", *Foundations and Trends in Electronic Design Automation*, Vol. 3, No. 1, pp. 1–95, 2008.

26. "IEC 62531:2012(E) (IEEE Std 1850-2010): Standard for Property Specification Language (PSL)", *IEC 62531:2012(E) (IEEE Std 1850-2010)*, pp. 1–184, Jun. 2012.

27. "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language", pp. 1–1315, Feb. 2018.

28. Sutton, R. S. and A. G. Barto, *Reinforcement Learning: An Introduction*, Adaptive Computation and Machine Learning Series, A Bradford Book, Cambridge, MA, USA, 2 edn., Nov. 2018.

29. Groce, A., "Coverage Rewarded: Test Input Generation via Adaptation-based Programming", *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 380–383, Nov. 2011.

30. Kim, J., M. Kwon and S. Yoo, "Generating Test Input with Deep Reinforcement Learning", *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*, pp. 51–58, May 2018.

31. Reddy, S., C. Lemieux, R. Padhye and K. Sen, "Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning", *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1410–1421, ACM, Seoul

South Korea, Jun. 2020.

32. Koroglu, Y., A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi and Y. Donmez, "QBE: QLearning-Based Exploration of Android Applications", *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 105–115, IEEE, Vasteras, Apr. 2018.

33. Koroglu, Y. and A. Sen, "Functional Test Generation from UI Test Scenarios using Reinforcement Learning for Android Applications", *Software Testing, Verification and Reliability*, Vol. 31, No. 3, p. e1752, 2021.

34. Singh, K., R. Gupta, V. Gupta, A. Fayyazi, M. Pedram and S. Nazarian, "A Hybrid Framework for Functional Verification using Reinforcement Learning and Deep Learning", *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, pp. 367–370, Association for Computing Machinery, New York, NY, USA, May 2019.

35. Wu, N., Y. Xie and C. Hao, "IRONMAN: GNN-assisted Design Space Exploration in High-Level Synthesis via Reinforcement Learning", *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, pp. 39–44, Association for Computing Machinery, New York, NY, USA, Jun. 2021.

36. Fedar, E., "Fedar F1", `https://github.com/eminfedar/fedar-f1-rv64im`, accessed in December 2021.

37. Snyder, W., "Veripool", `https://www.veripool.org/verilator/`, accessed in December 2021.

38. "Specifications", `https://riscv.org/technical/specifications/`, accessed in November 2021.