MAGCS: Multi-Agent Guided Configuration Search for Optimization Fault Detection in Logic Synthesis

Peiyu Zou[a], Xiaochen Li[a,*], Shikai Guo[b,*], Weihong Sun[c], Yuyao Xu[c], He Jiang[a]
[a]School of Software, Dalian University of Technology, Dalian, China
[b]School of Information Science and Technology, Dalian Maritime University, Dalian, China
[c]Hi-Think Technology, Corp, Dalian, China
{zoupeiyu, xiaochen.li}@mail.dlut.edu.cn, shikai.guo@dlmu.edu.cn, {sunwh, yuyao.xu}@dhc.com.cn, jianghe@mail.dlut.edu.cn

***Abstract*—Logic synthesis tools are crucial to translate high-level descriptions into optimized gate-level netlists. However, complex optimization operations and operation configurations can cause synthesis faults. To address this, we propose MAGCS, a fault detection method using multi-agent reinforcement learning to dynamically refine optimization sequences. MAGCS consists of three components: a test program selector that applies feature extraction and cosine similarity to curate diverse test programs, an optimization selector using the A2C algorithm to adaptively adjust operations and configurations, and an optimization fault verifier performing equivalence checks to pinpoint optimization-induced faults. Using MAGCS, we identified 32 confirmed faults on Vivado and Yosys, all of which are resolved. MAGCS received recognition from the Vivado community for its significant contributions to tool improvement[1].**

***Index Terms*—Logic synthesis tools, optimization sequences, optimization faults**

I. Introduction

FPGA and ASIC are widely used in complex hardware system design such as communications and aerospace [1, 2]. In this process, logic synthesis is a critical step, which transforms high-level hardware description languages (e.g., Verilog and VHDL) into optimized gate-level netlists, to improve the performance and power efficiency of the design [3]. As shown in Fig. 1, the logic synthesis process typically consists of four main stages: Translation, Constrain, Logic Optimization, and Mapping, each involving specific optimization operations and configurations. By adjusting these operations and configurations, developers can create customized optimization sequences to achieve expression simplification, timing optimization, and efficient resource allocation. However, certain optimization sequences may produce unintended results or design errors, such as increased delays from timing optimizations and the unintended removal of critical logic during simplification. This risk is particularly concerning in safety-critical domains.

Fig. 2 is an example of circuit design code for FPGA and ASIC, which operates registers using the clock signal clk and generates a 768-bit output signal y. The logic synthesis uses Yosys with the sequence `opt_clean -purge; opt_expr -mux_bool; opt_dff -nodffe; opt_reduce -full; opt_demorgan`. In this sequence, `opt_clean` and `opt_expr` are optimization operations, while `purge` and `mux_bool` are their configurations. An error occurred when using the `opt_demorgan` operation (Fault ID: 4610[2]). As indicated in the log in Fig. 2(b), the error is caused by a zero input signal to the `$reduce_or` unit, which triggers a `std::out_of_range` exception. The error occurred at line 11: `reg79 <= (~reg67[(4'h8):(3'h5)]);` Due to an incorrect signal connection, the input size was mistakenly set to zero, leading to synthesis failure. The Yosys community resolved this issue through a fix (Pull Request ID: 4612[3]), as illustrated in Fig. 2(c). The fix involved modifying the *opt_demorgan.cc* optimization file. A conditional check was added at lines 43-44 to prevent further processing when the input signal is zero.

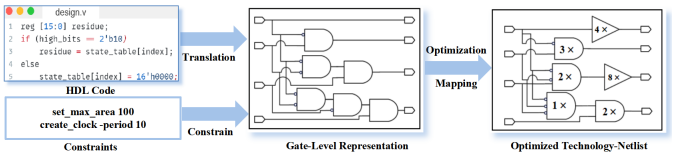


Fig. 1: Electronic Circuit Design - Logic Synthesis Flowchart.

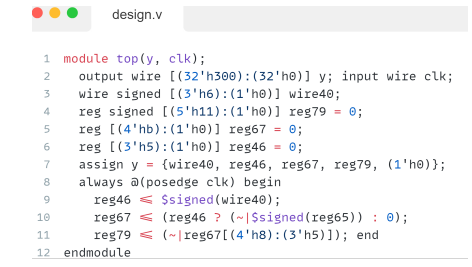
To identify faults in logic synthesis tools, several testing methods have been proposed [4, 5, 6, 7]. These methods typically rely on generators (e.g., Verismith and Vloghammer) to produce large numbers of Verilog files, which are then used as input (aka, test programs) to expose faults in synthesis tools. However, these methods often overlook the complexity introduced by user-defined optimization sequences. Methods like DeLoSo[8] address this by using heuristics to explore specific combinations of operations and parameter configurations that may trigger faults. However, DeLoSo lacks real-time adaptation based on feedback, limiting its exploration of broader optimization sequences. Additionally, DeLoSo relies on code generators to produce a single batch of test programs, which tend to exhibit similar structures within the batch, lacking sufficient diversity to comprehensively expose faults. Based on the current research landscape, fault detection in logic synthesis optimization faces two main challenges:

Challenge 1: Insufficient diversity in test programs. Fault detection relies on diverse test programs to thoroughly test the whole logic synthesis process. Limited variation among test programs restricts the ability of existing methods to reveal potential faults.

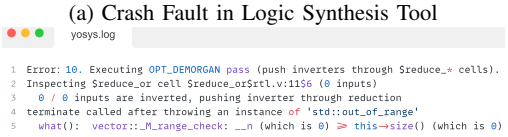
Challenge 2: Inadequate exploration of optimization sequences. Logic synthesis tools offer numerous optimization operations; for instance, Vivado has over 18 optimization operations, each with more than 10 possible operation configurations. Since only certain optimization sequences could be faulty, the vast search space of optimization sequences (e.g., over 10^{18+} for Vivado) makes exhaustive testing impractical.

To this end, we propose MAGCS, a multi-agent reinforcement learning-based method for detecting optimization faults in logic synthesis tools. To tackle the first challenge, MAGCS includes a test program selector component that uses circuit features to vectorize different test programs. MAGCS selects test programs with the greatest variability in terms of vector distances to ensure the

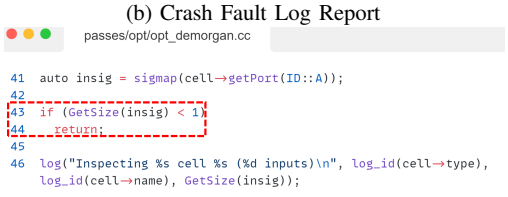
*Corresponding author (Xiaochen Li and Shikai Guo)
[1]https://support.xilinx.com/s/feed/0D54U00008Wfd2cSAB
[2]https://github.com/YosysHQ/Yosys/issues/4610
[3]https://github.com/YosysHQ/yosys/pull/4612



(a) Crash Fault in Logic Synthesis Tool



(b) Crash Fault Log Report



(c) Code Maintenance Patch

Fig. 2: Yosys Crash Fault (Fault ID: 4610)

diversity of test programs. To tackle the second challenge, MAGCS treats each optimization operation as an independent agent. MAGCS first determines the position of each optimization operation within the optimization sequence, then selects the corresponding operation configuration. Through multi-agent reinforcement learning, MAGCS dynamically adjusts the agents based on a reward function that prioritizes the fault detection ability of optimization sequences and minimizes equivalence check delays. MAGCS iteratively refines the optimization sequence to find optimal configurations for fault detection. Finally, an optimization fault verifier is used to determine if the optimized, synthesized design aligns with the intended functionality of the original design with equivalence check. If the equivalence is broken, an optimization fault could be found.

Experiments show that MAGCS outperforms baselines (i.e., DeLoSo, DynSwarm, and InitSwarm) in detecting optimization faults, with a fault detection rate 68.42% to 3100% higher than these methods. Over one month of testing on the commercial tool Vivado and the open-source tool Yosys, MAGCS identified 32 confirmed faults, demonstrating its effectiveness across different synthesis tools.

The contribution of this work includes: (1) the first multi-agent reinforcement learning-based approach for efficiently detecting optimization faults in logic synthesis tools; (2) 32 faults are found in Vivado and Yosys, all of which are confirmed and fixed; (3) the Vivado community acknowledges the value of our reported faults, recognizing our contribution to their tool improvement; (4) we have open-sourced MAGCS on Github[4].

II. RELATED WORK

Fault detection in logic synthesis tools is critical in FPGA and ASIC. Several automated testing frameworks have been proposed to generate diverse Verilog programs for fault detection. Herklotz et al. [4] developed Verismith, a tool that uses randomization techniques to generate complex Verilog programs for testing logic synthesis tools. Then, Vloghammer[5] was proposed, which can test the stability of Yosys. Ratchev et al.[6] developed VERGEN, which evaluates synthesis tool performance by generating structured random programs with specific design features. Thakur et al.[7] developed VeriGen, which uses template-based generation to test logic synthesis tools. While these approaches have been instrumental in fault detection, their primary focus is code generation; they cannot thoroughly explore faults arising from optimization sequences.

For optimization fault detection, Jiang et al. [8] proposed DeLoSo, which utilizes a heuristic method to explore optimization sequences that trigger faults in synthesis tool's optimization operations. However, DeLoSo faces limitations, such as being prone to local optima and not fully addressing test program diversity.

Overall, existing approaches lack thorough testing of optimization operations and operation configuration. This study effectively combines diverse optimization sequences with varied test programs to improve the effectiveness of fault detection.

III. MAGCS FRAMEWORK

To detect optimization faults in logic synthesis tools, we propose MAGCS, a multi-agent reinforcement learning-based fault detection method. As shown in Fig. 3, MAGCS has three core components to collaboratively select and validate optimization sequences.

A. Test Program Selector

To build a diverse and representative set of test programs, MAGCS employs a selection method based on feature extraction and similarity metrics. Given an initial set P of n_{seed} Verilog test programs, MAGCS represents each test program as a feature vector to encompass a diverse range of logic structures and timing characteristics.

Specifically, previous studies [9] have proposed five categories of features to represent a Verilog program, including

- *data processing and operations*, such as the number of arithmetic, logical, and comparison operators, which characterize data operation complexity;
- *data flow control and representation*, such as the number of assignment statements, variable declarations, and numerical representations, reflecting the synchronization and complexity of data flow;
- *structuring and modularization*, such as the number of module instantiation and declarations, describing the modular design traits of Verilog programs;
- *control flow and logic*, such as the number of conditional statements and loop control, which characterize the logical control and decision-making in Verilog programs;
- *abstraction and reuse*, such as the number of parameter definitions and scope identifiers, which capture the handling of parameterization and module reuse in Verilog programs.

We further add the sixth category of feature, namely *timing features*, which analyzes the characteristics of the timing behavior of hardware circuits, including the number of clock signals, flip-flops, edge-triggered logic, and non-blocking assignments in Verilog programs. These timing features are important for describing timing control in FPGA and ASIC designs, particularly in timing-sensitive circuits, enabling a more comprehensive characterization of Verilog program behavior.

Based on these features, MAGCS converts each test program p_i in P into a feature vector $F_i = [f_{i1}, f_{i2}, ..., f_{im}]$, where m represents the number of features. Each dimension in F_i is a quantified feature, e.g., f_{i1} counts the number of arithmetic operators in p_i, and f_{i2}

[4]https://github.com/MAGCS-method/MAGCS

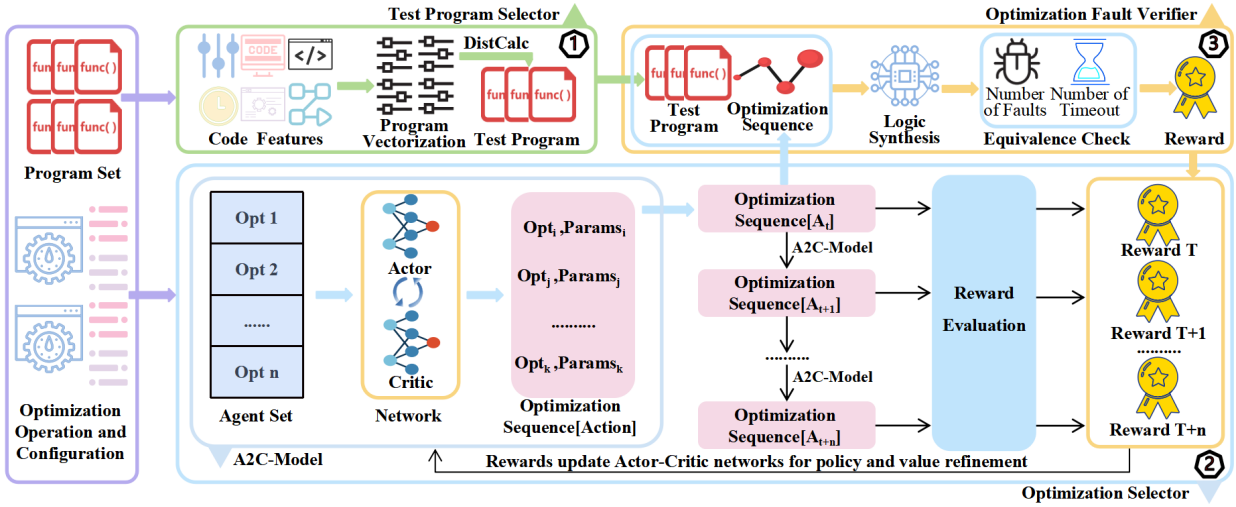


Fig. 3: The framework of MAGCS

is the number of logical operators in p_i. Once the feature vector is constructed, MAGCS normalizes each feature dimension j by:

$$\hat{f}_{ij} = \frac{f_{ij} - \min(f_{1j}, \ldots, f_{nj})}{\max(f_{1j}, \ldots, f_{nj}) - \min(f_{1j}, \ldots, f_{nj})} \quad (1)$$

where f_{ij} is the original feature value of test program p_i in dimension j, and $\min(f_{1j}, \ldots, f_{nj})$ and $\max(f_{1j}, \ldots, f_{nj})$ are the minimum and maximum values of dimension j across all n test programs.

After normalization, the feature vector of p_i is expressed as: $\hat{F}_i = [\hat{f}_{i1}, \hat{f}_{i2}, ..., \hat{f}_{im}]$. This normalization compresses the feature values of each dimension into the [0, 1] range, ensuring that all dimensions are on the same scale and avoiding bias.

Next, MAGCS uses cosine distance to measure the dissimilarity between test program features. Cosine distance reflects the angular difference between two feature vectors and is defined as: $d\left(\hat{F}_i, \hat{F}_j\right) = 1 - \frac{\hat{F}_i \hat{F}_j}{\left\|\hat{F}_i\right\| \left\|\hat{F}_j\right\|}$, where $\hat{F}_i \hat{F}_j$ is the inner product of the feature vectors, and $\left\|\hat{F}_i\right\|$ and $\left\|\hat{F}_j\right\|$ is the magnitudes of the vectors. A large $d\left(\hat{F}_i, \hat{F}_j\right)$ means two test programs are more dissimilar in the feature space.

To select a maximally diverse test program set P_{sel}, MAGCS employs a recursive greedy method. First, a test program p_1 is randomly selected from the initial set P to serve as the first program in P_{sel}. For each remaining test program $p_i \in P$, we compute the sum of its cosine distances to all already-selected programs in P_{sel}:

$$D\left(p_i\right) = \sum_{p_j \in P_{sel}} d\left(\hat{F}_i, \hat{F}_j\right) \quad (2)$$

Next, the test program p_k with the greatest dissimilarity to the current programs in P_{sel} is selected, satisfying the following condition:

$$p_k = \underset{p_i \in P / P_{sel}}{\arg\max}\ D\left(p_i\right) \quad (3)$$

Using this method, we iteratively build a diverse subset P_{sel} from P until it includes n_{sel} test programs. This method maximizes the diversity within the feature space, facilitating broader fault detection across various optimization scenarios in logic synthesis tools.

B. Optimization Selector

During the logic synthesis optimization process, the selection of optimization operations and their configurations creates a vast search space. For instance, in Yosys, commonly used optimization operations include `opt_expr`, `opt_clean`, `opt_merge`, `opt_muxtree`, and `opt_demorgan`. Each of them has various operation configuration. For example, `opt_expr` provides configurations such as `opt_expr -mux_bool` for converting Boolean expressions into multiplexers and `opt_expr -full` for applying comprehensive logic transformations. The execution order of these operations (i.e., the optimization sequence) directly affects the quality of the synthesis optimization.

To navigate this complex space, the optimization selector leverages a multi-agent reinforcement learning framework A2C [10] for optimizing the selection of operations and their configurations. In A2C, agents make decisions by selecting actions from an action space, which represents possible optimization operations and configurations, based on the current state of the system. The state space contains all relevant system information at a given time. By continuously learning from feedback rewards, agents optimize their decisions, identifying the best sequence of operations to improve fault detection. Let the state space S_t represent the optimization sequence and its corresponding operation configuration at a given time step t. S_t is composed of a set of optimization operations and their configurations:

$$S_t = \{(opt_{1,t}, params_{1,t}), \ldots (opt_{i,t}, params_{i,t}), ..., (opt_{n,t}, params_{n,t})\} \quad (4)$$

where $opt_{i,t}$ denotes the selection of the i-th optimization operation at time t, and $params_{i,t}$ represents its operation configuration. For instance, at time step t, the state might be:

$$S_t = \{(opt\_expr, fine), (opt\_clean, purge), (opt\_merge, nomux)\} \quad (5)$$

The action space A defines the decisions that the agent can make at each time step. Each agent's actions include selecting the operation configuration of the current optimization operation as well as dynamically adjusting the order of these operations. Specifically, action $a_t \in A$ consists of two parts: $opt_{i,t+1}$, which selects the next optimization operation, and $params_{i,t+1}$, which specifies the configuration for $opt_{i,t+1}$. Hence, the action a_t is expressed as:

$$a_t = \{(opt_{i,t+1}, params_{i,t+1})\} \quad (6)$$

To guide the agent in selecting effective actions in this complex optimization space, we designed a reward function that maximizes the fault detection ability of each action a_t while minimizing unnecessary

equivalence check timeouts. Specifically, the reward function R for an action a_t is defined as:

$$R = \theta \cdot \frac{N_{fault}}{N_{fault}+1} - (1-\theta) \cdot \frac{N_{timeout}}{N_{timeout}+1} \tag{7}$$

where N_{fault} represents the number of faults detected when applying the current optimization sequence on all test programs in P_{sel}, and $N_{timeout}$ represents the number of timeouts encountered during equivalence check for the same sequence and test set. Parameter $\theta \in [0, 1]$ controls the balance between positive rewards and negative penalties. Detecting an optimization fault results in a positive reward, encouraging the agent to explore and find more faults, thereby improving the effectiveness of the optimization sequence. Timeouts during equivalence check are penalized because they delay verification of the design against the netlist and disrupt the overall testing process. By using timeouts as a negative reward, MAGCS reduces unnecessary delays to ensure test efficiency.

Our framework adopts an Actor-Critic (A2C) approach with two primary networks: a policy network (Actor) and a value network (Critic) [11]. The policy network learns to select optimization actions (operations and configurations) that maximize cumulative rewards, while the value network estimates the expected reward value of the current state to enhance decision accuracy.

At each time step t, the agent uses the policy network to select an optimization action a_t based on the current state s_t, specifying the next optimization operation and its configuration. Executing a_t generates a new state s_{t+1}, reflecting the updated sequence of optimization operations and configurations.

After each action, an immediate reward R is calculated, evaluating the optimization sequence based on detected faults and equivalence check timeouts. The critic network estimates the expected value of the state to guide further decisions. A2C iteratively refines the agent's policy, enabling effective selection of optimization operations and configurations. The objective is to maximize the cumulative discounted reward:

$$\max_{\omega} E\left[\sum_{t=0}^{T} \gamma^t R_t\right] \tag{8}$$

where ϖ is the policy network parameter, $\gamma \in [0, 1]$ is the discount factor that reduces the influence of rewards in future time steps, and R_t is the immediate reward at time step t. Based on prior studies [10, 11], we set ϖ to 0.001 and γ to 0.99.

Through iterative training, the agent gradually optimizes its strategy, making optimal decisions in the complex optimization space and maximizing the cumulative reward. The optimization selector continuously improves fault detection by dynamically adjusting the optimization operations and configurations, ensuring that each stage of testing delivers high fault detection capacity and efficiency.

C. Optimization Fault Verifier

After identifying each optimization sequence with the optimization selector, it is converted into optimization instructions applied to the design file—a Verilog or HDL circuit representation—during synthesis. In some cases, specific test programs and optimization sequences may cause unexpected synthesis terminations, as discussed in Fig. 2. This type of optimization fault indicates that certain operations or configurations within the optimization process contain issues, directly causing the synthesis to fail. Feedback on detected faults and timeouts from each sequence is then relayed back to the optimization selector, helping refine its strategy.

For optimization sequences that complete synthesis successfully, the system generates synthesis file and conducts equivalence check to ensure that the synthesized logic matches the original design's functionality. The equivalence check uses formal verification tools to convert both the original design and the synthesized files into comparable logic expressions, verifying whether their outputs match across all input conditions. Specifically, the core of the equivalence check is to ensure that the output logic of the source design L_{src} matches the synthesized design L_{syn} for all inputs x:

$$\forall x \in X, L_{src}(x) = L_{syn}(x) \tag{9}$$

Here, X represents the complete set of possible inputs. If under certain input conditions, $L_{src}(x) \neq L_{syn}(x)$, this indicates that a functional error occurred during the synthesis process.

By monitoring the synthesis process and conducting equivalence check, it is possible to determine whether the optimization sequence led to faults. If the synthesis process terminates unexpectedly or if the equivalence check fails, this signifies an issue with the optimization process. The optimization fault verifier ensures that any errors introduced during optimization are detected.

IV. EVALUATION

We evaluated the effectiveness of MAGCS by answering the following research questions (RQs): RQ1 – Effectiveness of MAGCS Compared to Baseline Methods, RQ2 – Impact of Parameter on MAGCS, and RQ3 – Effectiveness of MAGCS in Finding Real Faults.

A. Experimental Setup

Hardware Environment: Our experiments were conducted on an x86_64 computer running Ubuntu 18.04.01, equipped with an Intel Core i9-12900 CPU @ 2.80GHz with 24 cores and 128GB RAM.

Software and Test Program Generation: Our experiments employed two mainstream logic synthesis tools: the commercial tool Vivado (version 2024.1) and the open-source tool Yosys (version 0.41+126). We chose the latest versions, as they are more likely to contain recent fault fixes, which are particularly critical for developers [8]. To generate the initial test set P, we used Verismith [4] for its ability to produce diverse and complex Verilog programs essential for fault detection. Following configurations from prior studies [8], we generated an initial set of $n_{\text{seed}} = 200,000$ programs, from which a greedy method selected a diverse subset of $n_{\text{sel}} = 1,000$ programs.

Types of Errors: We identified four types of optimization faults: (1) Crash Fault: a crash during synthesis with specific optimization sequence; (2) Performance Fault: a hang or stall in the synthesis process; (3) Parsing Fault: errors in parsing the design files; and (4) Logic Fault: discrepancies between the synthesized and original design due to logic errors.

B. Baseline Methods and Testing Strategy

To assess the performance of MAGCS, we used nine methods for comparison. These methods are based on four basic methods, i.e., Default, InitSwarm, DynSwarm, and DeLoSo [8]. Each method has two steps, i.e., test program selection and optimization sequence generation. For test program selection, we employed two strategies: *Rand*, where 1,000 test programs are randomly selected from a pool of 200,000, and *Div*, where 1,000 diverse test programs are selected using the test program selector proposed in Section 3.1.

For optimization sequence generation, we have:

Default: Default optimization sequences in synthesis tools (e.g., opt-fast and opt-full in Yosys, and O1, O2, O3 in Vivado) are used.

InitSwarm: Using the default optimization operations in their fixed order, with only the operation configuration randomized.

TABLE I: Effectiveness of MAGCS in Fault Detection

Tools	Default		InitSwarm		DynSwarm		DeLoSo		MAGCS	MAGCS
	Rand	Div	Rand	Div	Rand	Div	Rand	Div	Rand	
Vivado	0	0	1	1	2	3	7	9	16	26
Yosys	0	0	0	1	1	1	2	3	3	6
Total	0	0	1	2	3	4	9	12	19	32

DynSwarm: The positions of optimization operations are randomly rearranged, and operation configurations are also randomly assigned.

DeLoSo: Genetic algorithm is used to explore optimization sequences by adjusting the positions of optimization operations and tuning their configurations, aiming to identify combinations that may trigger faults.

When combining the two steps, we have eight method variants, i.e., Default-Rand, Default-Div, InitSwarm-Rand, InitSwarm-Div, DynSwarm-Rand, DynSwarm-Div, DeLoSo-Rand, and DeLoSo-Div. For example, Default-Rand means that we use the default optimization sequence with the randomly selected 1,000 test programs to test synthesis tools. Additionally, we create a variant of MAGCS as the ninth baseline, i.e., *MAGCS-Rand*, where the test program selector in MAGCS is replace with the Rand strategy.

Each testing method was allocated a fixed runtime. During each iteration, methods like MAGCS dynamically generated new optimization sequences, while Default used predefined or randomized configurations. These optimization sequences were applied for logic synthesis. We analyzed both the synthesis process and checked for equivalence between the synthesized netlist and the original design to identify potential faults. When mismatches were detected, functional simulations were performed to gather more detailed data, enabling easier replication of the faults by the community.

For each detected fault, failed assertions and logs were analyzed to remove duplicates. The Verilog program that triggers the faults was incrementally reduced to its minimal form for reproduction. The simplified design, the triggering optimization sequence, and relevant error logs were submitted to the development team for verification as either new or known issues.

C. RQ1: Effectiveness of MAGCS and Baselines

To answer RQ1, we use MAGCS and nine baseline methods to test logic synthesis tools for over one month. This period includes Verilog test program generation, test case selection and optimization selector with each method, and fault verification. This evaluation period aligns with previous studies [8]. For the tests, we used the latest versions of Vivado (2024.1) and Yosys (0.41+126). We use two identical machines to run all methods.

As shown in TABLE I, MAGCS identified 26 faults in Vivado and 6 in Yosys, significantly outperforming other methods. In comparison, DeLoSo-Div found 9 faults in Vivado and 3 in Yosys, totaling 12. DynSwarm-Div detected 4 faults, while InitSwarm performed poorly. Default-Rand and Default-Div did not find any faults, as the default optimization sequence are typically extensively tested before release. MAGCS demonstrated superior fault detection ability across different tools by dynamically optimizing sequence combinations through multi-agent reinforcement learning.

We also used a Venn diagram to further analyze the differences between MAGCS and other baselines in fault detection. Since Default and InitSwarm-Rand did not detect any faults in the experiment, they were excluded from the Venn diagram. As shown in Fig. 4, the faults found by MAGCS is a superset of those found by other baselines. This indicates that MAGCS can both effectively select diverse test programs and explore a broader optimization configuration space, uncovering faults missed by other methods.

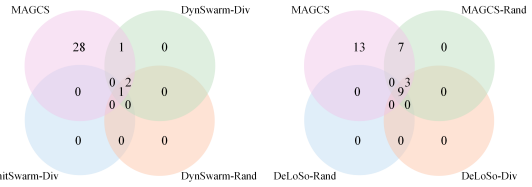


Fig. 4: Relationship of faults found by different methods.

TABLE II: Faults Detected by MAGCS with Different θ Values

Value of θ	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Vivado Faults	1	2	4	3	7	8	11	13	11	9	5
Yosys Faults	0	1	1	1	2	3	4	4	3	1	1
Total	1	3	5	5	9	11	15	17	15	12	6

Conclusion: MAGCS is highly effective in detecting optimization faults in logic synthesis tools, surpassing baseline methods by triggering more optimization faults. The faults found by MAGCS form a superset of those detected by other baselines.

D. RQ2: Impact of Parameter on MAGCS

To answer RQ2, we explore a key parameter in MAGCS: the coefficient θ in the reward calculation formula (Equation 7). This coefficient balances the detection of optimization faults with the penalty for timeouts during equivalence check. In the experiment, the value of θ ranged from 0.0 to 1.0, with intervals of 0.1. Notably, when $\theta = 0.0$, the system only penalizes timeouts without incentivizing fault detection. In contrast, when $\theta = 1.0$, the system entirely ignores timeouts and focuses solely on the number of detected optimization faults.

As shown in TABLE II, we recorded the number of faults detected by MAGCS in Vivado and Yosys for varying values of θ. As θ increased from 0.0 to 0.7, the number of detected faults steadily rose. In this range, MAGCS prioritized fault detection, while allowing some timeouts, which helped reveal additional faults. However, as θ exceeded 0.7, the lack of penalty for timeouts began to interfere with testing, causing verification disruptions and fewer detected faults. At $\theta = 0.7$, MAGCS achieved the best balance between fault detection and time management, with high detection efficiency.

Conclusion: When θ is set to 0.7, MAGCS achieves the best performance in detecting optimization faults in logic synthesis tools.

TABLE III: Details of Faults Found by MAGCS

ID	Tools	Fault ID	Type	Title
1	Vivado	8TyRQBSA3	CF	Vivado Crash in HARTHOptPost::prepDsps()
2	Vivado	8gjjsGSAQ	CF	Vivado Crash in HARTLOptAbc::runNlOpt()
3	Vivado	8cMtrMSAS	CF	HARTNlOptimize::modOptimize() Error in Vivado
4	Vivado	7BrmgwSAB	CF	Crash in HARTHOptPost::optimize() During Synthesis
5	Vivado	8boSFLSA2	CF	NDup::copyModule() Causes Vivado Setback
6	Vivado	8a5wHySAI	CF	HARTSWorker::runInternal() Crashing Vivado
7	Vivado	8YNb4PSAT	CF	Vivado Obstructed by hdi::tcltasks() Crash
8	Vivado	8jW2ngSAC	CF	HARTSWorker::runJob() Causing Vivado Failure
9	Vivado	8bGuVzSAK	CF	Vivado Crash in ConstProp::cleanup()
10	Vivado	8gjjtHSAQ	CF	Stack Overflow Check Causes Vivado Issues
11	Vivado	8gjjv8SAA	CF	Vivado Crash in GDpGen::implementBinary()
12	Vivado	8jVeCmSAK	CF	Crash Due to HARTTUpdateTNInstC::updateCell()
13	Vivado	8jW2nfSAC	CF	Optimize1::optimize() Malfunction in Vivado
14	Vivado	8aRj9KSAS	CF	HSynMod::connectInputPin() Crash in Vivado
15	Vivado	8jW2xgSAC	CF	Crash in NDbC::uniquePrefixes() in Vivado
16	Vivado	7AD9ZWSA1	LF	unsigned() Function Error Due to Synthesis Parameters
17	Vivado	8X0u8WSAR	PFF	Vivado Freezes During Ubuntu Synthesis
18	Vivado	8ZY2lqSAD	PFF	Vivado Optimization Termination During Synthesis
19	Vivado	8Vzsu9SAB	PFF	Large Design Causes Vivado to Freeze
20	Vivado	8aRjBGSA0	PFF	Ubuntu Design Optimization Freezes Vivado
21	Vivado	8boBgSSAU	PFF	Vivado Stalls During Synthesis on Ubuntu
22	Vivado	8bGuWxSAK	PFF	Vivado Hangs on Specific Verilog File During Synthesis
23	Vivado	8gjjwGSAQ	PFF	Synthesis Hang Issue in Vivado
24	Vivado	8XGDKYSA5	PFF	Vivado Stalls on Specific File in Optimization
25	Vivado	8YmcY9SAJ	PFF	Vivado Optimization Causes Process Hang
26	Vivado	8XVl5vSAD	PFF	Ubuntu Synthesis Process Hangs in Vivado
27	Yosys	4610	CF	Yosys Synthesis std::out_of_range Error
28	Yosys	4486	LF	Yosys Optimization Causes Incorrect Output
29	Yosys	4491	LF	Custom Yosys Passes Cause Faulty Synthesis
30	Yosys	4478	LF	Yosys Optimization Error in PEEPOPT Pass
31	Yosys	4427	PF	Yosys Verilog Parsing Error After File Read
32	Yosys	4458	PFF	Yosys Synthesis Hash Table Overflow

Note: Crash faults (CF), Logic faults (LF), Parsing faults (PF), Performance faults (PFF).

Vivado.log

```
# An unexpected error has occurred (11)
Stack:
/lib/x86_64-linux-gnu/libc.so.6(+0x42520) [0x7fcb9d242520]
/Vivado/2024.1/lib/lnx64.o/libtcl8.6.so(TclNRRunCallbacks+0x47) [0x7fcb9d484497]
/Vivado/2024.1/lib/lnx64.o/librdi_commontasks.so(+0x25c6d1) [0x7fcb9245c6d1]]
/Vivado/2024.1/lib/lnx64.o/librdi_synth.so(HARTSWorker::runJob(HARTSWorkerConfig&)+0x3f)
/Vivado/2024.1/lib/lnx64.o/librdi_synth.so(HARTSWorker::mainLoop()+0x8b) [0x7fcb64d5bc4b]
/Vivado/2024.1/lib/lnx64.o/librdi_synth.so(HARTNDb::parallelListen(char const*)
/Vivado/2024.1/lib/lnx64.o/librdi_synth.so(+0x14a4a74) [0x7fcb642a4a74]
/Vivado/2024.1/lib/lnx64.o/librdi_synth.so(+0x13623e5) [0x7fcb641623e5]
/Vivado/2024.1/lib/lnx64.o/libtcl8.6.so(TclNRRunCallbacks+0x47) [0x7fcb9d484497]
```

Fig. 5: Crash Fault Log Report for Fault ID: 8jW2ngSAC

(a) Netlist after correct synthesis by the logic synthesis tool

(b) Erroneous netlist resulting from a logic synthesis tool fault

Fig. 6: Comparison of waveform differences

E. RQ3: Effectiveness in Finding Real Faults

To answer RQ3, we used MAGCS to detect optimization faults in Vivado and Yosys. As shown in TABLE III, we have reported a total of 32 optimization faults to the Vivado and Yosys development teams or communities, including 16 crash faults, 11 performance issues, 1 parsing fault, and 4 logic faults. All reported faults have been confirmed. We also shared the details of these fault on GitHub. Below, we provide an example for each category of faults.

Crash Fault: Vivado Fault ID: 8jW2ngSAC[5] occurred due to improper resource scheduling during complex optimization operations. As shown in the crash log in Fig. 5, the crash happened while executing a synthesis task. The Vivado thread scheduling mechanism failed to properly manage multi-task loads, particularly for high-complexity tasks like hierarchy flattening and clock-gating conversion. The log indicates that the functions $HARTSWorker :: runJob()$ and $HARTSWorker :: runInternal()$ encountered synchronization issues during parallel task execution, leading to resource contention. Additionally, the $parallelListen()$ function failed to properly schedule parallel tasks, causing thread blocking and ultimately triggering the crash.

Logic Fault: Vivado Fault ID: 7AD9ZWSA1[6] is the inconsistency of simulation waveform between the original design and the synthesized netlist. At the 159ps timestamp in the waveform, as shown in the red box in Fig. 6, a clear discrepancy is evident. The first (correctly synthesized) netlist shows a value of 802, while the second (incorrectly synthesized) netlist shows a value of 800. This significant behavioral inconsistency issue arose in the module responsible for logic optimization during synthesis. Logic optimization failed to maintain the timing consistency of the design when dealing with certain synchronous or nested logic. Although subtle, this timing discrepancy can seriously cause the overall functionality of the circuit to fail, especially in high-speed designs.

Parsing Fault: Yosys Fault ID:4427[7], as shown in Fig. 7, indicates that Yosys failed to continue execution while generating the RTLIL representation due to an assertion failure. The assertion failure occurred at $frontend/ast/ast.cc : 855$, where $node- > bits == v$ failed. In this case, Yosys encountered a bit-width mismatch when generating the RTLIL representation for the modules top and

[5]https://adaptivesupport.amd.com/s/question/0D54U00008jW2ngSAC
[6]https://support.xilinx.com/s/question/0D54U00007AD9ZWSA1
[7]https://github.com/YosysHQ/Yosys/issues/4427

yosys.log

```
Generating RTLIL representation for module \top
Generating RTLIL representation for module \module282
ERROR: Assert 'node→bits ==v' failed in frontends/ast/ast.cc:855.
```

Fig. 7: Parsing Fault Log Report for Fault ID: 4427

yosys.log

```
"terminate called after throwing an instance of 'std::length_error'
what(): hash table exceeded maximum size."
```

(a) Log Report

```
kernel/hashlib.h
@@ -188,6 +188,7 @@ inline unsigned int mkhash(const T &v) {
188 188
189 189   inline int hashtable_size(int min_size)
190 190   {
    191 +       // Primes as generated by https://oeis.org/A175953
191 192         static std::vector<int> zero_and_some_primes = {
192 193                 0, 23, 29, 37, 47, 59, 79, 101, 127, 163, 211, 269, 337, 431, 541, 677,
193 194                 853, 1069, 1361, 1709, 2137, 2677, 3347, 4201, 5261, 6577, 8231, 10289,
@@ -196,7 +197,9 @@ inline int hashtable_size(int min_size)
196 197                 897133, 1121423, 1401791, 1752239, 2190299, 2737937, 3422429, 4278037,
197 198                 5347553, 6684443, 8355563, 10444457, 13055587, 16319519, 20399411,
198 199                 25499291, 31874149, 39842687, 49803361, 62254207, 77817767, 97272239,
199     -               121590311, 151987889, 189984863, 237481091, 296851369, 371064217
    200 +               121590311, 151987889, 189984863, 237481091, 296851369, 371064217,
    201 +               463830313, 579787991, 724735009, 905918777, 1132398479, 1415498113,
    202 +               1769372713
200 203         };
```

(b) Code Maintenance Patch

Fig. 8: Yosys Performance Fault (Fault ID: 4458)

$module282$. The error occurred in the frontend Abstract Syntax Tree parsing phase, where Yosys has a signal width mismatch during the conversion from Verilog program to RTLIL.

Performance Fault: Yosys Fault ID:4458[8] was triggered during logic synthesis optimization compilation. The hash table size exceeded the system's allowable limit, triggering a $std :: length\_error$ exception, as shown in Fig. 8(a). This issue arose primarily because Yosys did not handle hash table size calculations properly when processing large design files, leading to rapid hash table expansion during optimization. As shown in Fig. 8(b), Yosys developers resolved this issue[9] by adjusting the hash table size calculation logic, using a larger prime number table to allocate hash table capacity. This ensured that the hash table would function correctly when handling large designs.

Conclusion: MAGCS demonstrates excellent capabilities in detecting optimization faults in logic synthesis tools. MAGCS successfully identifies 32 optimization faults, which are all confirmed and fixed.

V. Conclusion

Optimization faults in logic synthesis tools can critically impact hardware design accuracy and performance; therefore, effective fault detection is essential. We present MAGCS, a reinforcement learning-based multi-agent method to enhance fault detection in logic synthesis optimizations. Tested on Vivado and Yosys, MAGCS demonstrated high fault detection capabilities, finding 32 confirmed faults in both tools. The Vivado community acknowledged our fault reports, and shared a note of gratitude[10]: *"Thank you for reporting a series of Vivado synthesis issues and providing the related example designs. Most of the issues can be reproduced and have been reported to AMD's developers to improve our tool."*.

VI. Acknowledgment

This work was supported by Key Research and Development Project of Liaoning Province under Grant No. 2024JH2/102400059, and in part by the National Natural Science Foundation of China under Grant No. 62202079, No. 62032004, No. 62472062.

[8]https://github.com/YosysHQ/Yosys/issues/4458
[9]https://github.com/YosysHQ/Yosys/pull/4471
[10]https://support.xilinx.com/s/feed/0D54U00008Wfd2cSAB

REFERENCES

[1] G. Yan, X. Liu, and H. Wang, "Fast fpga accelerator of graph cut algorithm with out-of-order parallel execution in folding grid architecture," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[2] G. Brilli, G. Valente, A. Capotondi, P. Burgio, T. Di Masciov, and Valente, "Fine-grained qos control via tightly-coupled bandwidth monitoring and regulation for fpga-based heterogeneous socs," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[3] L. Witschen, T. Wiersema, L. Reuter, and M. Platzner, "Search space characterization for approximate logic synthesis," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 433–438.

[4] Y. Herklotz and J. Wickerson, "Finding and understanding bugs in fpga synthesis tools," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 277–287.

[5] C. Wolf, "VlogHammer https://github.com/YosysHQ/VlogHammer," 2019.

[6] B. Ratchev, M. Hutton, G. Baeckler, and B. van Antwerpen, "Verifying the correctness of fpga logic synthesis algorithms," in *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, 2003, pp. 127–135.

[7] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 3, pp. 1–31, 2024.

[8] H. Jiang, P. Zou, X. Li, z. Zhou, and S. Guo, "Deloso: Detecting logic synthesis optimization faults based on configuration diversity," *ACM Transactions on Design Automation of Electronic Systems*, 2024.

[9] "Losyte https://github.com/LoSyTe-Logic-Synthesis-Tool-Test/LoSyTe," 2024.

[10] V. Mnih, "Asynchronous methods for deep reinforcement learning," *arXiv preprint arXiv:1602.01783*, 2016.

[11] J. Lu, J. Yang, S. Li, Y. Li, W. Jiang, and J. Dai, "A2c-drl: Dynamic scheduling for stochastic edge-cloud environments using a2c and deep reinforcement learning," *IEEE Internet of Things Journal*, 2024.