Simulink Compiler Testing via Configuration Diversification with Reinforcement Learning

Xiaochen Li, Shikai Guo, Hongyi Cheng, and He Jiang, *Member, IEEE,*

***Abstract*—Simulink compiler testing is important since all cyber physical system (CPS) models are required to be compiled by Simulink compiler. Current testing processes use CPS models generated by CPS model generators for testing. Since the effectiveness of CPS model generators heavily relies on suitable generator configurations, existing approaches randomize configurations or infer configurations with historical bug information to generate diverse bug-triggering CPS models. However, these approaches are designed for general-purpose compilers (e.g., GCC), which have two challenges when testing Simulink compiler, namely the *CPS model representation challenge* on representing CPS models for diversity measurement and the *configuration learning* challenge on learning configurations to generate diverse CPS models. To address these challenges, we propose RECORD, a new configuration diversification approach. RECORD has a feature vectorization component, which addresses the first challenge by representing CPS models as feature vectors to capture the local and global characteristics of CPS models for diversity measurement. RECORD then uses a reinforcement learning component to generate diverse CPS models based on the learned relationship between configuration updates and diversity changes, thus addressing the second challenge. Experiments demonstrate that within three months, RECORD reported 11 confirmed Simulink compiler bugs, significantly outperforming the state-of-the-art configuration diversification approaches. RECORD can also facilitate different testing strategies to find more bugs.**

***Index Terms*—Cyber-physical system, Simulink, configuration diversification, compiler testing, reinforcement learning**

I. Introduction

CYBER physical system (CPS) development tools, such as MathsWork Simulink, are fundamental platforms for developers to design and analyze a CPS before developing embedded code [1], [2], [3]. In a CPS development tool, developers design CPS models, a kind of block diagram with blocks and connections, to simulate the behavior of a CPS. As an industrial standard [2], Simulink has been widely used by developers to design CPS models for many safety-critical applications, such as aerospace and healthcare [4], [5], [6], [7]. Simulink analyzes and compiles a CPS model with its compilation system (i.e., Simulink compiler), and then generates embedded code for deploying in target CPS applications. However, Simulink compiler may contain bugs. These bugs can inject unexpected behaviors into a CPS model, which heavily threaten the correctness and safety of target CPS applications.

To find Simulink compiler bugs, several approaches for Simulink compiler testing have been proposed [1], [2], [3]. These approaches conduct testing based on a large number of CPS models generated by CPS model generators (e.g., SLforge [1]). The CPS models are compiled by Simulink compiler with different settings that are expected to obtain the same outputs. By analyzing the output inconsistency, Simulink compiler bugs can be found based on the idea of differential testing [1]. In addition, these CPS models can also be mutated to generate equivalent CPS model variants for Equivalence Modulo Input (EMI)-based testing [2].

Despite the importance of automatically generated CPS models, CPS model generators largely depend on suitable configurations (e.g, the probability to generate a certain type of blocks) to generate bug-triggering CPS models, since the same configuration tends to generate CPS models with a similar block distribution and trigger duplicate bugs [8]. In the area of general-purpose compiler testing (e.g., GCC and LLVM), two approaches have been proposed to generate diverse programs by configuration diversification, including swarm testing [9] and history-based testing [8]. They randomize configurations or use historical bug-triggering programs to infer configurations for generating diverse programs to explore more compiler input space.

However, swarm testing and history-based testing may not test Simulink compiler effectively due to the CPS model representation challenge and the configuration learning challenge. The first challenge occurs, because these studies generate programs by controlling the probability to generate different local program characteristics (i.e., features), such as *return* statements. However, most of these features are not supported by CPS model generators. They also ignore the value of global CPS model characteristics (e.g., the CPS model architecture) for guiding CPS model generation. Hence, it is important to mine effective features to represent CPS models for measuring their diversity. Regarding the second challenge, history-based testing relies on thousands of programs that can trigger bugs in old compiler versions (generated by the program generator Csmith) to infer configurations. However, as a kind of well-developed commercial software, it is not feasible for a CPS model generator to find thousands of bugs in Simulink compiler for configuration learning. Hence, the second challenge is to learn the configurations to generate CPS models with different block distributions, without using historical bug-triggering CPS models.

To generate diverse CPS models effectively, we propose

X. Li and H. Jiang are with the School of Software, Dalian University of Technology, Dalian, China, and Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province. H. Jiang is also with DUT Artificial Intelligence Institute, Dalian, China. E-mail: xiaochen.li@dlut.edu.cn, jianghe@dlut.edu.cn (corresponding author)

S. Guo and H. Cheng are with the School of Information Science and Technology, Dalian Maritime University, Dalian, China. E-mail: shikai.guo@dlmu.edu.cn, chenghongyi@dlmu.edu.cn

Manuscript received August XX, 2022.

RECORD, a **R**einforcement l**E**arning based **CO**nfigu**R**ation **D**iversification approach for Simulink compiler testing, to address the aforementioned challenges. RECORD has two main components, i.e., a feature vectorization component and a reinforcement learning component. RECORD aims to generate CPS models with diverse distributions of bug-triggering features by diversifying CPS model generator configurations. These CPS models are expected to explore larger input space of Simulink compiler to trigger more bugs. In the feature vectorization component, we construct a set of local and global features, which capture the characteristics of both a single block in a CPS model (e.g., the number of *Subsystem* blocks) and a complete CPS model (e.g., the CPS model architecture). We vectorize CPS models as feature vectors for measuring their diversity to address the first challenge. The reinforcement learning component iteratively generates CPS models with different CPS model generator configurations. In each iteration, this component learns the reward (i.e., diversity) to be got when certain changes are made to configurations by comparing the feature vectors of newly and previously generated CPS models. With this knowledge, RECORD trains a double deep Q-network (DDQN) to intelligently update configurations for generating CPS models with high expected rewards. Hence, the configuration learning challenge is addressed. Finally, CPS models generated in each iteration are used to test Simulink compiler by differential testing.

To evaluate the effectiveness of RECORD, we compare different configuration diversification approaches for Simulink compile testing. RECORD significantly outperforms the state-of-the-art approaches (i.e., swarm testing and history-guided testing) in terms of the bug-finding capability of Simulink compiler. Within three months, RECORD found 11 confirmed bugs in the recent Simulink version R2021b, including 9 new bugs. In contrast, baselines only found at most three bugs. CPS models generated by RECORD can also be mutated to facilitate other compiler testing strategies. Using these CPS models, an EMI-based Simulink compiler testing approach SLEMI finds 3 more bugs compared to using CPS models generated by a single set of configurations. In addition, by analyzing the distribution of different blocks in the generated CPS models, we find that the reinforcement learning component improves the ability of RECORD to generate diverse CPS models.

In summary, this study makes the following contributions:

- We propose a novel approach RECORD to test Simulink compiler. RECORD combines a feature vectorization component and a reinforcement learning component to address the CPS model representation and the configuration learning challenges in Simulink compiler testing, respectively.
- We conduct extensive experiments to assess the bug-finding capability of RECORD. RECORD found 11 confirmed Simulink compiler bugs, including 9 new bugs.
- We release RECORD as a tool for Simulink compiler testing [10].

The rest of the paper is organized as follows. Section II presents the background and explains the motivation and challenges of this study. Section III introduces the basic machine learning algorithms used in this paper. Section IV describes the core component of RECORD. Section V reports on the evaluation of RECORD. Section VI-C discusses the threats to validity. Section VII surveys related work. Section VIII concludes the paper.

TABLE I
CATEGORIES OF BLOCKS IN THE CPS MODELING LANGUAGE

#	Library	Description
1	Continuous	Continuous function blocks such as *Derivative* and *Integrator*
2	Customizable Blocks	Blocks with customizable appearance that control parameter values and display signal values during simulation
3	Dashboard	Blocks that can control parameter values and display signal values during simulation
4	Discontinuities	Discontinuous function blocks such as *Saturation*
5	Discrete	Discrete time function blocks such as*Unit Delay*
6	Logic and Bit Operations	Logic or bit operation blocks such as *Logical Operator* and *Relational Operator*
7	Lookup Tables	Lookup table blocks such as *Cosine*
8	Math Operations	Mathematical function blocks such as *Sum*
9	Matrix Operations	Blocks for modeling matrix operations
10	Messages & Events	Blocks for modeling message-based communication
11	Model Verification	Blocks for self-verifying models, such as *Check Input Resolution*
12	Model-Wide Utilities	Model-wide operation blocks such as *Model Info* and *Block Support Table*
13	Ports and Subsystems	Blocks related to subsystems, such as *Inport*, *Outport*, *Subsystem*, and *Model*
14	Signal Attributes	Modify signal attribute blocks such as *Data Type Conversio*n
15	Signal Routing	Route signal blocks such as *Bus Creator*
16	Sinks	Display or export signal data blocks such as *Scope* and *To Workspace*
17	Sources	Generate or import signal data blocks such as *Sine Wave* and *From Workspace*
18	String	String manipulation blocks
19	User-Defined Functions	Custom function blocks such as *MATLAB Function* and *MATLAB System*

II. BACKGROUND AND MOTIVATION

A. CPS Models and Simulink

In a CPS development tool, developers design a CPS with CPS models. As a commercial CPS development tool, Simulink has become an industry standard, which is used to design and analyze CPS models for many safety-critical applications, such as aerospace and healthcare [4], [5], [6], [7]. Simulink analyzes and compiles CPS models with its compilation system (i.e., Simulink compiler), and then executes the compiled CPS models to simulate the behavior of the CPS. When the simulation is the same as developers' expectations, embedded code is generated with Simulink, which is finally deployed in the target CPS.

A CPS model is designed by the CPS modeling language. Each CPS model is a block diagram that contains a set of blocks and their connections. A block performs some operations on the data (which are usually signals) received from its input ports, and passes the computation results to the output ports [1], [2], [11]. These output ports are connected to

subsequent blocks through a set of connections (i.e., lines in the block diagram) for further computation. At last, the final computation results are presented to developers.

As shown in Table I, there are 19 categories of blocks in the CPS modeling language classified by Simulink developers [12]. Each category is organized as a library, which includes several blocks to reflect some characteristics of CPS models. For example, the *Ports and Subsystems* library contains blocks related to such as *Inport*, *Outport*, *Subsystem*, and *Model*. Blocks in this library can define hierarchical subsystem structures of CPS models, where a parent block contains several child blocks with the acyclic relation. However, the CPS modeling language does not have complete and publicly-available formal language specifications [13], [14], [15]. Language semantics are usually defined in the closed-source and complex code base of Simulink, which increases the difficulty to design and generate valid CPS models.

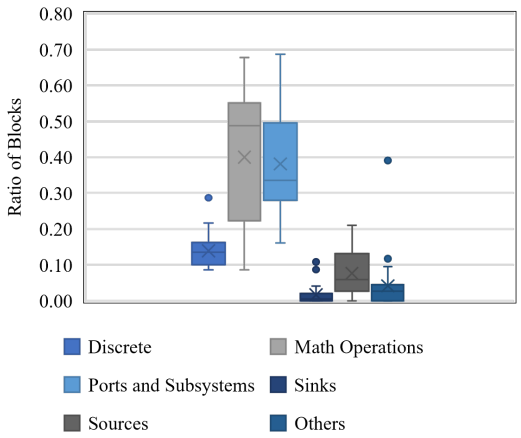

Fig. 1. Distribution of blocks in the collected CPS models

B. Simulink Compiler Testing by CPS Model Generation

Although all CPS models are analyzed and compiled through Simulink compiler, Simulink compiler may contain bugs [1], [2]. These bugs can crash Simulink or inject unexpected behaviors in target CPS applications.

Existing approaches for Simulink compiler testing are based on differential testing. These approaches take CPS models as input. The CPS models are compiled by Simulink compiler with different settings that are expected to obtain the same outputs. If the outputs are different, a Simulink compiler bug could be triggered. Since differential testing requires a large number of CPS models, several CPS model generators have been proposed (e.g., CyFuzz [3] and SLforge [1]). Currently, SLforge is the state-of-the-art CPS model generator [1], [2]. SLforge supports a subset of the most-used CPS modeling language specifications, such as specifications for *Discrete*, *Math Operations*, *Ports and Subsystems*, *Sinks*, and *Sources* libraries [1]. SLforge has a configuration file to guide the CPS model generation, which defines the probability to generate blocks in each library, the structure of CPS models (e.g., the maximum level of the hierarchy, the number of blocks), and the execution strategy (e.g., the threshold for time-out, the simulation mode). Based on the configuration, SLforge generates CPS models. These CPS models can be used for classical differential testing or regarded as inputs (i.e., seed CPS models) for other testing strategies (e.g., EMI-based testing [2]). Studies showed that SLforge had triggered 8 new Simulink compiler bugs in five months [1].

C. Motivation

CPS model generators largely depend on suitable configurations (e.g., the probability to generate blocks in each library) to generate bug-triggering CPS models. However, existing studies for general-purpose compilers (e.g., GCC and LLVM) show that program elements (e.g., *if* and *return* statements in C) have diverse probabilities to trigger compiler bugs [16], [8]. Since few studies investigate this phenomenon in the CPS modeling language, we conduct a preliminary study to analyze the characteristics of bug-triggering CPS models to motivate this study.

We manually reproduced 50 CPS models that can trigger Simulink compiler bugs from the MathWorks' bug reporting platform [17]. We collected 50 CPS models for two reasons. On the one hand, the complete list of bug reports is neither available nor traceable on this platform. Only a few Simulink compiler bugs are available to be collected and reproduced. On the other hand, as a preliminary study, 50 bugs can already show the distribution of different program elements (i.e., blocks) in bug-triggering CPS models.

Fig. 1 presents the distribution of blocks in the most-used libraries [1], which are also the libraries supported by SLforge. We add a class 'others' to count the distribution of blocks in other libraries. We find CPS models that can trigger Simulink compiler bugs have diverse block distributions. *Math Operations* and *Ports and Subsystems* are the two most important libraries to find Simulink compiler bugs. However, the variance of block distributions is also large. For some CPS models, the ratio of blocks in these two libraries is less than 0.2. Hence, the diversity and the interaction of blocks in different libraries should be considered when generating CPS models. Such diversity cannot be handled by CPS model generators using a single set of configurations.

These findings demonstrate that it is important to configure CPS model generators to generate CPS models with diverse distributions of bug-triggering blocks to effectively test Simulink compiler.

D. Challenges

To configure program generators, there are two main approaches in the literature. Both of them are for general-purpose compiler testing.

- Swarm testing [9], which generates diverse test programs by randomizing generator configurations
- History-guided testing [8], which uses historical bug-triggering programs to infer the range of each configuration option. It then uses particle swarm optimization

(PSO) to iteratively optimize the inferred range with newly generated programs.

However, these approaches may not test Simulink compiler effectively due to two challenges, i.e., the CPS model representation challenge and the configuration learning challenge.

CPS model representation challenge. Programming languages supported by general-purpose compilers usually have complete language specifications. For example, existing approaches use 71 configuration options in the C program generator Csmith as features to control the generated programs, such as the probability to generate the *const* keyword and *return* statements. However, most of these features are not supported or controllable by CPS model generators. In addition, these features only focus on local program characteristics. They ignore global CPS model characteristics such as the size and complexity of CPS models. Global characteristics reflect the impact of a combination of different blocks, which are also important for generating bug-triggering CPS models. The first challenge is *how to mine effective features to represent CPS models for measuring their diversity*.

Configuration learning challenge. Existing approaches rely on a large number of historical programs to infer the range of each configuration option. For example, history-guided testing collects more than 4000 bug-triggering programs generated by Csmith to guide the program generation. However, such historical knowledge is not available in Simulink compiler, because the complete list of Simulink compiler bugs is not traceable in the MathWorks' bug reporting platform. Moreover, Simulink compiler testing is more challenging. It is difficult to generate thousands of bug-triggering CPS models by simply running CPS model generators (e.g., SLforge). The second challenge is *how to learn configurations of CPS model generators to generate CPS models with diverse features*.

III. Preliminary

Before illustrating our approach, we first briefly introduce two basic machine learning techniques we rely on.

A. Reinforcement Learning

Reinforcement learning [18] is a class of machine learning algorithms that take actions in an environment to maximize the notion of cumulative return. There are four basic concepts in reinforcement learning.

- *State (S)* is the state of environment. For example, in a chess game, the state can be the current position of pieces on the chess board.
- *Action (A)* is the operation conducted on the state, such as moving one of the pieces in the chess board.
- *Reward (R)* is the immediate reward obtained after taking actions on states ($\mathbf{S}\times\mathbf{A}\rightarrow\mathbf{R}$). For example, if function f represents the probability to win the chess game at state s_t, the immediate reward $r_t = f(s_{t+1}) - f(s)$, where $s_{t+1} = a_t(s_t)$ and a_t is the action on s_t.
- *Cumulative return (G)* is the overall reward for an action. $g_t = r_t + \gamma r_{t+1} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$, where γ is a factor to discount the impact of an action on future rewards.

Reinforcement learning aims to train a model (without needing labeled input/output pairs) to learn the cumulative reward can be got when certain actions are made on the state based on the "feedback" of the immediate reward. The model is then used to predict the action that should be taken at a state.

B. Double Deep Q-Network (DDQN)

DDQN [19] is one of the state-of-the-art algorithms for training the reinforcement learning model. The kernel of DDQN includes two deep neural networks with the same architecture, denoted as Q_ϕ and $TargetQ_{\widehat{\phi}}$, where ϕ and $\widehat{\phi}$ represent the parameters (i.e., weights of edges and nodes) of the two networks. Q_ϕ is used to train and predict actions, while $TargetQ_{\widehat{\phi}}$ rectifies Q_ϕ to reduce the fluctuations in the training process. DDQN aims to optimize ϕ for Q_ϕ to accurately predict the cumulative return of each action.

The execution of DDQN is iterative. Initially, ϕ is randomly initialized, and $\widehat{\phi} = \phi$. In each iteration t, based on the current state s_t, the ε-greedy strategy [20] is used to select an action.

$$a_t = \begin{cases} random(A),\ p(\varepsilon) \\ \arg\max Q_\phi(s_t, a),\ p(1-\varepsilon) \end{cases}, \tag{1}$$

where A presents all the actions that can be taken on s_t. DDQN has a probability ε to randomly select an action and a probability $1-\varepsilon$ to decide actions by Q_ϕ. For the latter, DDQN uses Q_ϕ to predict the cumulative reward obtained for each action $a \in A$, and selects action a_t having the highest predicted cumulative reward. When a_t is determined, DDQN calculates the actual immediate reward r_t for moving from s_t to s_{t+1} based on a predefined function f. DDQN uses an experience pool P to save r_t as a tuple $\langle s_t, a_t, s_{t+1}, r_t\rangle$.

In each iteration, DDQN selects a subset of tuples in P to optimize ϕ. Specifically, DDQN assumes that the cumulative return is only affected by the recent two immediate rewards, i.e., $g_t = r_t + \gamma r_{t+1}$. Given a tuple $\langle s_t, a_t, s_{t+1}, r_t\rangle$, the loss function for this tuple is

$$loss(\phi) = (g_t - Q_\phi(s_t, a_t))^2, \tag{2}$$

where $Q_\phi(s_t, a_t))$ is the cumulative return predicted by Q_ϕ. Here, g_t is defined as:

$$\begin{aligned} g_t &= r_t + \gamma r_{t+1} \\ &= r_t + \gamma\, TargetQ_{\widehat{\phi}}(s_{t+1}, a_{t+1}), \end{aligned} \tag{3}$$

where $a_{t+1} = \arg\max Q_\phi(s_{t+1}, a)$. It means DDQN first uses Q_ϕ to estimate the best action a_{t+1} at state s_{t+1}. Then, a_{t+1} is fed to $TargetQ_{\widehat{\phi}}$ to estimate the immediate reward r_{t+1}. Based on the *loss* of each tuple, parameters ϕ for Q_ϕ are updated in each iteration with gradient descent for better predicting g_t. Meanwhile, $\widehat{\phi}$ in $TargetQ_{\widehat{\phi}}$ is updated for every C iterations by copying parameters ϕ in Q to $\widehat{\phi}$. Since $TargetQ_{\widehat{\phi}}$ is updated less frequently than Q. $TargetQ_{\widehat{\phi}}$ can improve the stability of the optimizing process without being affected by the fluctuation in an iteration.

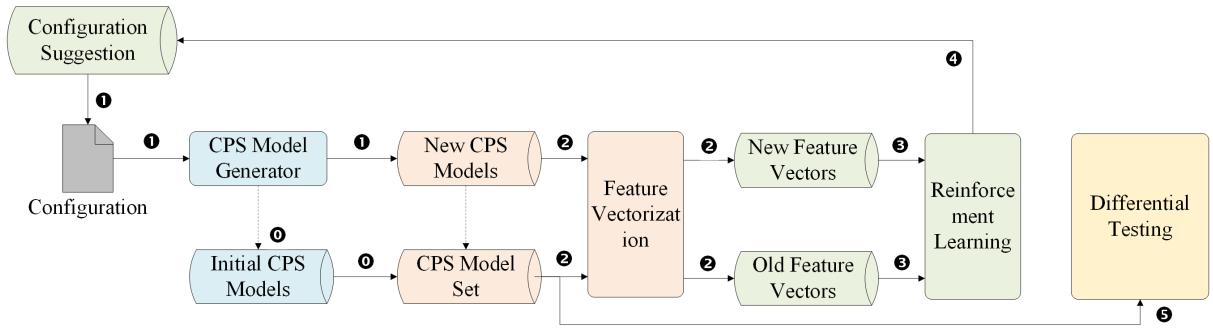

Fig. 2. The framework of RECORD

IV. The Framework of RECORD

In this section, we illustrate the framework of RECORD, a **R**einforcement l**E**arning based **CO**nfigu**R**ation **D**iversification approach for Simulink compiler testing. We first provide an overview of RECORD. We then explain the two main components of RECORD, i.e., feature vectorization and configuration learning.

A. Overview

The framework of RECORD is illustrated in Fig. 2. RECORD aims to generate CPS models with diverse distributions of bug-triggering features by configuration diversification of a CPS model generator. With diverse CPS models, larger input space of Simulink compiler can be tested, which is expected to trigger more Simulink compiler bugs. The inputs of RECORD are a configuration file of a CPS model generator (e.g., SLforge) and a group of CPS models generated by a set of random option values (Step 0). In each iteration, RECORD updates configurations and generates a group of new CPS models (Step 1). These CPS models are fed into the feature vectorization component, which vectorizes CPS models into feature vectors based on a set of manually-mined bug-triggering features (Step 2). By representing CPS models as feature vectors, the CPS model representation challenge is addressed. In the reinforcement learning component, RECORD compares feature vectors between new CPS models and previously generated CPS models. By comparison, RECORD uses reinforcement learning to learn the rewards (i.e., diversity) to be got when certain configuration updates are made to address the configuration learning challenge (Step 3). Based on the learned knowledge, this component suggests new configurations, which are expected to generate CPS models with diverse distributions of bug-triggering features (Step 4). At the end of each iteration, the newly generated CPS models are used to test Simulink compiler by differential testing (Step 5).

B. Feature Vectorization

In this component, RECORD represents a CPS model as a feature vector for measuring the diversity of CPS models. Table II presents the features used by RECORD, including the category and sub-category of features, the number of features, the description, the intuition to mine these features, and the value type. We construct these features by analyzing the semantics of blocks and the characteristics of bug-triggering CPS models collected in Section II-C. The key insight is that these features can reflect the capability of a CPS model to trigger Simulink compiler bugs. RECORD could explore more bug-prone space in Simulink compiler, if it generates CPS models with diverse distributions of these features.

Features. We construct two categories of features, including local features and global features. Local features capture the characteristic of a single block, while global features analyze blocks in a CPS model as a whole.

Specifically, as shown in Table II, we have four sub-categories of local features, including features related to *inputs and outputs*, *ports and subsystems*, *datatypes*, and *operations*. In each sub-category, there are several features. Each feature represents the frequency of a certain block in a CPS model. For example, there are 38 features related to *inputs and outputs*, which count the frequency of blocks such as *Display*, *Clock*, *Sine Wave*, *Constant*, *Scope*, *Terminator*, and *Outport* in a CPS model. We consider these blocks as features because they frequently interact with the outside world. The uncertainty of the outside world may explore some unexpected compilation branches in Simulink compiler. RECORD attempts to generate CPS models with different distributions of these blocks to thoroughly test Simulink compiler. The third and fourth columns in Table II explain the details of each sub-category. In total, we construct 113 local features.

Local features reflect the characteristics of certain blocks. However, as discussed in Section II-C, the interactions of blocks should be also considered when generating CPS models. To this end, we construct 4 sub-categories of global features to capture the characteristics of a single CPS model and a group of CPS models, including features related to the *size*, the *architecture*, and the *readability* of a CPS model, and the success rate for generating a group of CPS models. For example, a CPS model with low readability could mean that the CPS model has many *subsystems* and *operations* on *inputs and outputs* data. In global features, we add a feature *Success rate* to analyze the characteristic of a group of CPS models. Since the success rate for generating CPS models by a generator (e.g., SLforge) significantly changes when the generated CPS models are complex (e.g., with many blocks) [1], we vary the success rate to generate groups of CPS models with different levels of complexity.

TABLE II
FEATURES FOR CPS MODEL VECTORIZATION

Category	Sub-category (# of Features)	Description	Intuition	Value
Local Features	Inputs and outputs (38)	For a CPS model, we collect a subset of blocks in *Sources* and *Sinks* libraries, such as *Display*, *Clock*, *Scope*, *Record*, *Terminator*, *Ramp*, *Ground*, *Step* blocks. The frequency of each block is a feature.	These blocks frequently interact with I/O data (e.g., signal data) from/to the outside world. The uncertainty of inputs/outputs may explore more compilation branches.	Integer
	Ports and Subsystems (25)	For a CPS model, we collect a subset of blocks in the *Ports and Subsystems* library, such as *If*, *SwitchCase*, *Enable*, *For Each Subsystem*, *Function-Call Generator*, *Trigger*, *Switch Case*, *Subsystem* blocks. The frequency of each block is a feature.	CPS models with subsystems are usually more complex. They have a higher chance to trigger Simulink compiler bugs.	Integer
	Datatypes (14)	For a CPS model, we collect a subset of blocks in the *Signal Attributes* library, such as *Data Type Conversion*, *Bus to Vector*, *IC*, *Unit Conversion*, *Width*, *Rate Transition*, *Data Type Duplicate* blocks. The frequency of each block is a feature.	Data type conversion in CPS models is error-prone, which could throw exceptions such as *data type mismatch*. Simulink compiler may not correctly handle these problematic blocks.	Integer
	Operations (36)	For a CPS model, we collect a subset of blocks in the *Math Operations* library, such as *Add*, *MinMax*, *Gain*, *Product*, *Reshape*, *Sqrt*, *Dot Product*, *CompareToZero* blocks. The frequency of each block is a feature.	Blocks in this category can lead to complex math operations in CPS models. When optimizing these operations with Simulink compiler, bugs could occur.	Integer
Global Features	Size (5)	We calculate the size of a CPS model based on (a) the number of inputs and outputs in the CPS model; (b) the number of blocks in the CPS model; (c) the number of lines in the CPS model; (d) the number of subsystems in the CPS model; (e) the depth of the hierachical children of the CPS model. We take each number as a feature.	Larger CPS models usually have more block interactions, which increase the difficulty for Simulink compiler to optimize.	Integer
	Architecture (1)	We calculate the cyclomatic complexity of a CPS model.	Different CPS model architectures may trigger different optimization bugs in Simulink compiler.	Integer
	Readability (1)	We calculate the degree of data and structure layer separation.	The coupling of data and structure increases the difficulty to analyze CPS models by Simulink compiler.	Integer
	Success rate (1)	We collect CPS models generated by the CPS model generator in each iteration. We calculate the success rate of the generator.	The success rate can reflect the complexity of a group of CPS models.	Double

Vectorization. After feature construction, RECORD represents each CPS model as a feature vector, which is used to guide the generation of CPS model groups with diverse feature distributions. These CPS models could test the input space of Simulink compiler thoroughly.

In this study, we use $M = \{m_1, m_2, \ldots, m_i, \ldots, m_n\}$ to represent a group of CPS models generated by a CPS model generator with a certain combination of configuration option values, where n is the number of CPS models in the group, and m_i is the ith CPS model. The corresponding feature vectors of M are represented as $V = \{v_1, v_2, \ldots, v_i, \ldots, v_n\}$. We define v_i as the feature vector of the ith CPS model: $v_i = [x_{i1}\ x_{i2} \ldots x_{ij} \ldots x_{ir}]$, where r is the number of features and x_{ij} is the value of the jth feature. All the values in v_i are normalized by min-max normalization. Specifically, we use ${x_{ij}}^*$ to represent the original value (e.g., the frequency of a block) of x_{ij}, where $1 \leq i \leq n$ and $1 \leq j \leq r$. We normalize value ${x_{ij}}^*$ in feature vector v_i as:

$$x_{ij} = \frac{{x_{ij}}^* - \min\limits_{1 \leq i \leq n} ({x_{ij}}^*)}{\max\limits_{1 \leq i \leq n} ({x_{ij}}^*) - \min\limits_{1 \leq i \leq n} ({x_{ij}}^*)} \tag{4}$$

The feature vectorization component of RECORD addresses the CPS model representation challenge for Simulink compiler testing by representing CPS models as a group of feature vectors.

C. Configuration Learning

The reinforcement learning component learns the relationship between the configurations of a CPS model generator and the generated CPS models. This component aims to generate diverse CPS models by altering configurations. Since we represent CPS models as a group of feature vectors, we define *diversity* as the distance between different feature vector groups. Given two groups of CPS models M_1 and M_2, their feature vectors are V_1 and V_2, respectively. The center of a feature vector group $V = \{v_1, v_2 \ldots v_i \ldots v_n\}$ can be represented as $Center(V) = [x_1^c\ x_2^c \ldots x_j^c \ldots x_r^c]$, where $x_j^c = (\sum_{1 \leq i \leq n} x_{ij})/n$ is the mean of the jth feature in V. We follow the previous study [8] to define the distance of two CPS model groups as the Manhattan Distance of the centers of two feature vector groups: $d(M_1, M_2) = d(Center(V_1), Center(V_2)) = \sum_{j=1}^{j=r} |x_j^{c_1} - x_j^{c_2}|$.

Hence, the goal of RECORD is to configure a CPS model generator to generate new CPS model groups, which have a large distance from previously generated CPS model groups.

In this study, we generate diverse CPS model groups by reinforcement learning [18]. The basic idea of RECORD is that RECORD takes the distance between two CPS model groups as a reward. In each iteration, RECORD uses reinforcement learning (i.e., double deep Q-network (DDQN) [19] in this study) to learn the cumulative rewards can be got when certain actions (i.e., changes) are made on configurations, with

Algorithm 1 The Reinforcement Learning Component

Input: Configuration options O of a CPS model generator,
Parameters ϕ of the DDQN network Q_ϕ,
Action set on configurations A,
Number of episode N_{ep} and number of iterations N_{iter},
Threshold th for minimum ratio of valid CPS models.
Probability ε for the ε-greedy strategy.
Output: Set of generated CPS models $\mathcal{M}$;

1: $\mathcal{M} = \emptyset$;
2: Initialize an experience pool $P = \emptyset$;
3: **for** episode = 1 : N_{ep} **do**
4: Randomly initialize ϕ for Q_ϕ;
5: Randomly initialize configuration options O;
6: Generate CPS model group M_1 with O;
7: $\mathcal{M} = \mathcal{M} \cup M_1$;
8: Compute feature vectors V_1 for M_1;
9: Initialize a feature vector group set $\mathcal{V} = \{V_1\}$;
10: **for** t = 1 : N_{iter} **do**
11: Select action $a_t = \varepsilon(Q_\phi(V_t, A))$;
12: Update configuration option $O = a_t(O)$;
13: Generate CPS model group M_{t+1} with new O;
14: Compute feature vectors V_{t+1} for M_{t+1};
15: Compute reward r_t with V_{t+1} and $\mathcal{V}$;
16: Add tuple $\langle V_t, a_t, V_{t+1}, r_t \rangle$ into P;
17: Randomly select a subset of tuple $P' \in P$;
18: Update network $Q_\phi = train(Q_\phi, P', t)$;
19: $\mathcal{V} = \mathcal{V} \cup V_{t+1}$, $\mathcal{M} = \mathcal{M} \cup M_{t+1}$;
20: **if** $RatioOfValid(M_{t+1}) < th$ **then**
21: **break**;
22: **end if**
23: **end for**
24: **end for**

the generated CPS model groups in previous iterations and the corresponding rewards. With this knowledge, RECORD intelligently updates the configurations in each iteration to generate new CPS models that expect to obtain a high cumulative reward. Compared with swarm testing and history-guided testing, RECORD can learn to generate diverse CPS models without the knowledge of bug-triggering CPS models in history.

Based on the explanation of reinforcement learning in Section III, in this study we have:

- *State* is the values of configuration options. Given an option value combination, we can generate a group of CPS models M and calculate their feature vectors V;
- *Action* is the operation on configuration options (e.g., adding 0.02 to the value of a configuration option);
- *Reward* is the minimal distance between the newly generated CPS model group M_{t+1} and previously generated CPS model groups $M_1, \ldots, M_t$, i.e., $r_t = \min_{1 \le i \le t}(d(M_i, M_{t+1}))$;
- *Cumulative return* is $g_t = r_t + \gamma r_{t+1}$;

RECORD aims to update configurations to maximize g_t.

The pseudo-code of this component is depicted in Algorithm 1. Initially, RECORD initializes a set $\mathcal{M}$ to save the generated CPS models and an experience pool P (lines 1–2). RECORD conducts iteration for N_{ep} episodes. In each episode, RECORD randomly initializes the network Q_ϕ (line 4) and the configurations of the CPS model generator O (line 5). RECORD generates a group of CPS models with the initial configurations, and adds the corresponding feature vectors V_1 to a set $\mathcal{V}$ (lines 6–9).

After initialization, RECORD iteratively generates CPS models. In each iteration t, RECORD decides the actions on configuration options O based on the network Q_ϕ (line 11. We define three actions A on an option, including adding 0.02 to the value, reducing 0.02 on the value, and keeping the value unchanged. RECORD uses the ε-greedy strategy (Eq. 1) to select an action. Specifically, RECORD generates all the action combinations on all configuration options. Then, the network Q_ϕ is used to predict the cumulative reward for each action combination based on V_t. RECORD selects the action combination a_t which has the highest predicted cumulative reward. When a_t is decided, RECORD updates configurations in line 12.

With new configurations, a new group of CPS models M_{t+1} is generated (line 13). RECORD calculates the immediate reward r_t (i.e., diversity) obtained by actions a_t based on the feature vectors of M_{t+1} and groups of previously generated CPS models (saved in $\mathcal{V}$) (lines 14–15). We take r_t as "experience", which reflects the actual immediate reward to be got when feature vectors change from V_t to V_{t+1} by actions a_t. We save this experience as a tuple in P (line 16). In each iteration, we randomly select a subset of "experience" to update parameters ϕ of the network Q_ϕ using the standard DDQN optimization process explained in Section III-B (lines 17–18). Hence, RECORD can continuously improve its configuration learning ability as P increases in each iteration and episode.

RECORD saves the group of newly generated CPS models and their feature vectors in $\mathcal{M}$ and $\mathcal{V}$ respectively, where $\mathcal{M}$ is used for compiler testing and $\mathcal{V}$ is used for calculating the immediate reward in the next iteration. RECORD has two termination conditions to avoid repeatedly exploring local configuration space. When the number of iterations reaches N_{iter} or the ratio of valid CPS models in M_{t+1} is lower than a threshold th (line 20), we re-initialize RECORD for the next episode of iterations.

The output of configuration learning is the CPS models generated in each iteration for Simulink compiler testing. These CPS models are generated under the consideration of CPS model diversity.

V. Evaluation

In this section, we assess the effectiveness of RECORD for Simulink compiler testing. We first assess the bug-finding capability of the generated CPS models by RECORD compared against the state-of-the-art approaches. We then take the CPS models generated by RECORD as seeds to assess the ability of RECORD to facilitate different Simulink compiler testing strategies. At last, we analyze the impact of our configuration learning strategy on generating diverse CPS models.

Specifically, we aim to answer the following Research Questions (RQs).

RQ1 *How is the bug-finding capability of RECORD compared to the state-of-the-art approaches?*
RQ2 *Can we use the CPS models generated by RECORD as seeds to facilitate the EMI-based testing strategy?*
RQ3 *What is the impact of the reinforcement learning component on the diversity of generated CPS models?*

A. Baselines

We generate CPS models based on the CPS model generator SLforge [1], since it is the state-of-the-art generator for Simulink compiler testing. We consider three baselines:

- Default [1]. SLforge has default configurations suggested in the configuration file. This approach uses the default configurations to generate CPS models for testing.
- Swarm [9]. We use swarm testing to randomize the configurations of SLforge. In each iteration, a new set of random configurations is used to generate CPS models.
- History [8]. Chen et al. propose a history-guided testing approach, which uses PSO to optimize configurations of a generator based on the knowledge of historical bug-triggering programs. However, as explained in Section II-D, it is not feasible to infer the possible range of each configuration option with the historical knowledge for Simulink compiler. For this baseline, we only use PSO to optimize the configurations and generate a group of new CPS models in each iteration.

B. Testing Strategy

We run each approach for a given period of time. In each iteration, RECORD and baselines update the values of configuration options, which are used by SLforge to generate a group of new CPS models. We compile these CPS models with Simulink compiler, and find compiler bugs by differential testing. Specifically, we follow previous studies [1], [2] to compile each CPS model in both *Normal* and *Accelerator* simulation modes of Simulink compiler, where the former is the default simulation mode and the latter speeds up simulation by emitting native code. The two simulation modes are expected to obtain the same outputs. If the outputs are different, a Simulink compiler bug could be found.

When a possible bug is found, we compare the failed assertion and back-trace of this bug with the bugs found in previous iterations. We consider two bugs as duplicate bugs, if their failed assertions and back-traces are the same. After removing duplicate bugs, we conduct CPS model reduction. We remove blocks in the CPS model one by one, until a minimum bug-triggering CPS model is found. We submit each bug as a bug report to the MathWorks' bug reporting platform [17]. In the bug report, we present the symptom of the bug and the reduced CPS model that triggers the bug. The MathWorks support team provides two types of feedback for each bug, including *new* and *known*. *New* means the reported bug is a new one in Simulink compiler. *Known* means there is a duplicate bug in their repository with the reported bug. Since the complete bug list is not available in the MathWorks' bug reporting platform, we have published all the bugs found by RECORD in the replication package [10].

C. Implementation and Settings

We implement RECORD as a Matlab program. The reinforcement learning model DDQN is implemented with the Matlab Reinforcement Learning Toolbox [21].

As explained in Section II-A, SLforge has several configuration options, which sets the probability to generate blocks in a library, the structure of CPS models, and the execution strategy. In this study, we focus on optimizing seven default options related to block generation probability, including the probability to generate blocks in *Discrete*, *Math Operations*, *Ports and Subsystems*[1], *Sinks*, and *Sources*[2] [1] libraries, since they are the main options to affect the distribution of blocks in a CPS model. For other options, we use the default values in SLforge. For example, the hierarchy level is between 1 and 5, the number of blocks in a CPS model is between 30 and 100, and the timeout is 300 seconds. SLforge suggests these values because the time to generate a CPS model significantly increases as the hierarchy level and the number of blocks increase, leading to a frequent timeout. In each iteration, we generate 100 CPS models (i.e., the default value). All valid CPS models are used for differential testing.

For the reinforcement learning component, we set the number of iterations $N_{iter} = 10$ and the threshold $th = 0.10$. These two parameters are set empirically by running RECORD to test Simulink compiler for one week. Probability ε is set to $\varepsilon = power(0.1, \lceil episode/4 \rceil)$ according to existing studies [19]. This setting means that As the number of episodes increases, the value of ε (i.e., the probability to randomly select configurations) is decreasing, since RECORD has more experience in the experience pool to train the model for selecting actions. The discount factor γ for calculating the cumulative return is 0.9 [21]. The number of episodes N_{ep} is set dynamically according to the testing period.

We performed our experiments with computers running Windows 10 64-bit system with a 2.10 GHz Intel(R) Xeon(R) Silver 4166 processor and 128 GB of memory. The source code of RECORD and experiment results are available in our replication package [10].

D. Comparison with Baselines (RQ1)

In this RQ, we assess the effectiveness of RECORD compared to the baselines for finding Simulink compiler bugs.

1) Methodology: We run each approach with an evaluation period of three months, including the time to generate CPS models and conduct differential testing. The evaluation period is comparable with the previous study [1]. We choose the latest Simulink version when initializing this study for testing, i.e., Simulink R2021b. We choose this version because developers confirm and fix bugs primarily in the latest version [22]. These bugs are usually new and more important to developers since many known bugs have been fixed. In addition, after communicating with developers, they also suggested running CPS models on the new Simulink version. To accelerate the

[1]There are two options to separately control the probability to generate blocks related to *Subsystem* and *If* in this library.

[2]There are two options control the probability to generate *Constant* blocks and other blocks.

TABLE III
NUMBER OF BUGS FOUND BY RECORD AND BASELINES

	New	Known	Total
Default	1	0	1
Swarm	2	0	2
History	3	0	3
RECORD	9	2	11

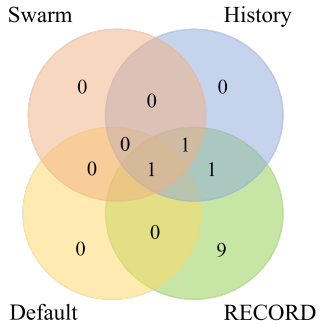

Fig. 3. Relationship of bugs found by different approaches

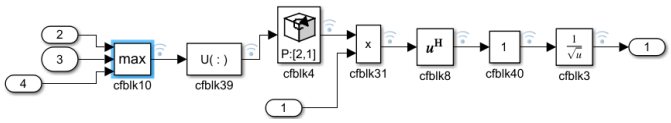

Fig. 4. TSC05290681: Inconsistent data in different simulation modes

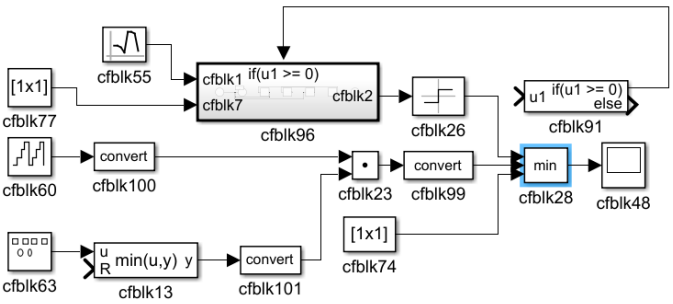

Fig. 5. TSC05313680: Min module mishandling NaN and 0 values

testing process, we follow the previous study [22] to test Simulink compiler in parallel. We use two computers with the same hardware configurations to run RECORD and the three baselines. The whole experiment took about six months.

We reproduce SLforge (*Default*), swarm testing (*Swarm*), and history-guided testing (*History*) based on the publicly available source code provided in previous studies [8], [1]. We have cross-reviewed the implementation and made necessary modifications (e.g., skipping the knowledge on bug-triggering programs used by *History*) for baselines.

2) Result: Table III shows the number of bugs found by RECORD and baselines. RECORD finds 11 confirmed bugs during the testing period, of which 9 bugs are *new* to developers and 2 bugs are labeled as *known*. Although Simulink is usually fully tested before release, RECORD can still find new bugs. Since Simulink is widely used for safety-critical applications [4], [5], [6], [7], finding these bugs is important to developers. RECORD significantly outperforms Default, Swarm, and History, which find 1, 2, and 3 bugs in this testing period, respectively.

The results obtained by different approaches can be explained in two folds. On the one hand, existing studies show that approaches such as SLforge found 8 bugs in five months on several Simulink versions from 2015a to 2017a [1]. However, most of the bugs could have been fixed in the latest versions. SLforge uses the same configurations to test Simulink compiler. The bug-finding capability tends to be saturated. On the other hand, although Swarm and History perform better than SLforge by changing configurations, the random and PSO strategies may not efficiently generate diverse CPS models for testing. In contrast, RECORD continuously learns the rewards obtained when updating different configuration options. With this knowledge, RECORD can explore more input space of Simulink compiler, which leads to a higher probability to trigger different Simulink compiler optimization bugs.

Fig. 3 shows the relationship of bugs found by different approaches. Since these approaches are built on top of the same CPS model generator, there is an overlap between the bugs found by different approaches. In particular, bugs found by RECORD are a super set of the other three approaches. It means RECORD can efficiently generate diverse CPS models to test Simulink compiler.

We list the detail of bugs found by RECORD in Table IV, including the ID (i.e., the Technical Support Case (TSC) ID), the title of the TSC, and the feedback from developers (FB). All bugs have been confirmed. RECORD finds Simulink compiler bugs related to data loss errors, math calculation errors, subsystem errors, prompt errors, and others. We classify these bugs based on their symptoms and the communication with developers. Here, we present some examples of (reduced) bug-triggering CPS models generated by RECORD. The original CPS models can be found in the replication package [10].

Data loss error: We find a bug (i.e., TSC05290681) related to the inconsistent outputs of the *Max* block in the *Normal* and *Accelerator* simulation modes. The CPS model is presented in Fig. 4. After debugging, the problem is that the *Max* block loses the signal data at the 7th second when optimizing the CPS model in the *Accelerator* simulation mode. Developers have reproduced this bug with the submitted CPS model. TSC05290678 is a duplicate bug of TSC05290681. The two bugs are being analyzed together by developers.

Math calculation error: Simulink compiler has a few bugs when optimizing math operations. The first type of bug is related to optimizing the boundary conditions of blocks. A typical example is TSC05313680. As shown in Fig. 5, in this CPS model, the *Min* block incorrectly compares the input values. When the inputs are NaN, 0, and -4.19e+8, the CPS model outputs -4.19e+8 in the *Normal* simulation model, while the output is NaN in the *Accelerator* simulation model. The reason is that Simulink compiler has problems to compare a negative value with NaN in the *Accelerator* simulation model. This type of bug is also found in *Max* (TSC05313899) and *Sum* (TSC05309988) blocks. Developers labeled each of these

TABLE IV
THE DETAIL OF BUGS FOUND BY RECORD

#	ID	Title	FB	Fixed
1	05246986	Data exceptions caused by different simulation operations	N	Y
2	05290678	Data loss and inconsistency in different simulation modes	K	Y
3	05290681	Inconsistent data in different simulation modes	N	Y
4	05309988	In accelerated simulation, abnormal data occurs in sum module	N	Y
5	05313680	MIN module mishandling nan and 0 values	N	Y
6	05313899	Max module acceleration simulation exception	N	Y
7	05372237	Unreasonable compilation error prompt	N	N
8	05385316	Model JIT accelerated simulation generate file failed	N	Y
9	05385318	UnitDelay module accelerator data exception in the if subsystem	N	Y
10	05405363	Models with if-action subsystems don't compile in accelerator	N	Y
11	05474416	Compilation fails due to zero-crossing detection in accelerator	K	Y

bugs as a new one. The second type is related to zero-crossing detection. We find the simulation of a CPS model could be terminated in the *Accelerator* simulation model due to the zero-crossing detection of the *MinMax* block. An instance of this bug is TSC05474416. Developers explained that they will fix this bug in the future Simulink version.

Subsystem error: Subsystems increase the complexity of CPS models. We find several bugs in Simulink compiler for subsystem optimization. For example, Fig. 6 is a CPS model with an *if* block (ID:cfblk114). When the *if* block is active, the *dot* block (ID:cfblk4) in the subsystem produces inconsistent results in different simulation modes. After debugging, we find that Simulink compiler generates a lot of unexpected code around the *dot* block in the *Accelerator* simulation mode. The code cannot be linked to any behaviors depicted in the CPS model. The bug has been forwarded to developers for further investigation. We have found several other bugs related to subsystems, such as TSC05385318 and TSC05405363

Wrong prompt: Simulink compiler can show inconsistent prompts during compilation. When we compile the CPS model in Fig. 7 in the *Accelerator* simulation mode, Simulink compiler shows a warning about "the datatype (unit32) passed to the *PID* block is not supported". However, we find the *PID* block can accept the datatype unit32. Developers suggested a temporal solution to add a *Data Type Converter* block in the CPS model. They created a request to improve the prompt.

Others: We label a bug as others because developers are still analyzing the root causes. For example, a CPS model[3] generated by RECORD reports an error "code generation assertion 'tidIdx < getNumTs()' failed in ..." in the *Accelerator* simulation mode (in Fig. 8). However, the CPS model can be compiled in the *Normal* simulation mode. Developers have confirmed the bug. They created a request to analyze the reason.

3) Importance of Bugs: We discuss the importance of the bugs by analyzing the type of reported bugs and the ratio of confirmed/fixed bugs. First, in the official bug reporting platform, developers classify bugs into four categories: security, incorrect code generation, assistive functionality issues, and others. For the bugs found by RECORD, most of them belong to incorrect code generation, which is listed by developers as an independent and main type of bugs in the platform. Such bugs can lead to incorrect simulation results that are different from the actual behavior of the designed CPS. In safety-critical domains such as medicine or aerospace, these bugs could inject unexpected behaviors into a CPS model, which heavily threaten the correctness and safety of target CPS applications.

Second, most of our reported bugs have been confirmed and fixed by developers. As shown in Table IV, nine of the reported bugs are confirmed as 'new' by developers. We determined whether a bug is fixed or not in two ways, i.e., the communication with developers and the replicability in a newer Simulink version. On the one hand, developers send us emails to confirm the resolution of a bug. On the other hand, we execute each bug-triggering CPS model on a newer version of Simulink. If the bug cannot be reproduced, it usually means that developers have fixed this bug. As shown in Table IV, the last column "Fixed" indicates whether a bug has been fixed ("Y") or not ("N"). For the reported bugs, ten of them have been fixed by developers, which illustrates the importance of these bugs (as it often takes substantial effort to fix a bug [35]).

Answer to RQ1: *RECORD significantly outperforms the baselines in terms of the bug-finding capability. RECORD finds 11 bugs in the testing period.*

E. Effectiveness of RECORD for EMI-based Testing (RQ2)

CPS models are the elementary inputs for Simulink compiler testing. In addition to finding bugs directly with the generated CPS models using differential testing, these CPS models are also important inputs for mutation-based compiler testing strategies (e.g., EMI-based testing). RQ2 assesses the effectiveness of the generated CPS models by RECORD for EMI-based Simulink compiler testing.

1) Methodology: In the literature, SLEMI [2] is the state-of-the-art EMI-based Simulink compiler testing strategy. SLEMI takes CPS models generated by a generator as seeds. SLEMI supports four mutation strategies on seed CPS models [2], including (1) randomly deleting blocks in zombie regions (i.e., blocks in unexecuted regions), (2) replacing zombie regions with *Saturation* blocks, (3) promoting blocks to their child model (as a subsystem), (4) and randomly creating a new signal path terminated by a sink block (which

[3]The reduced CPS model is too large to present. It is available in the replication package.

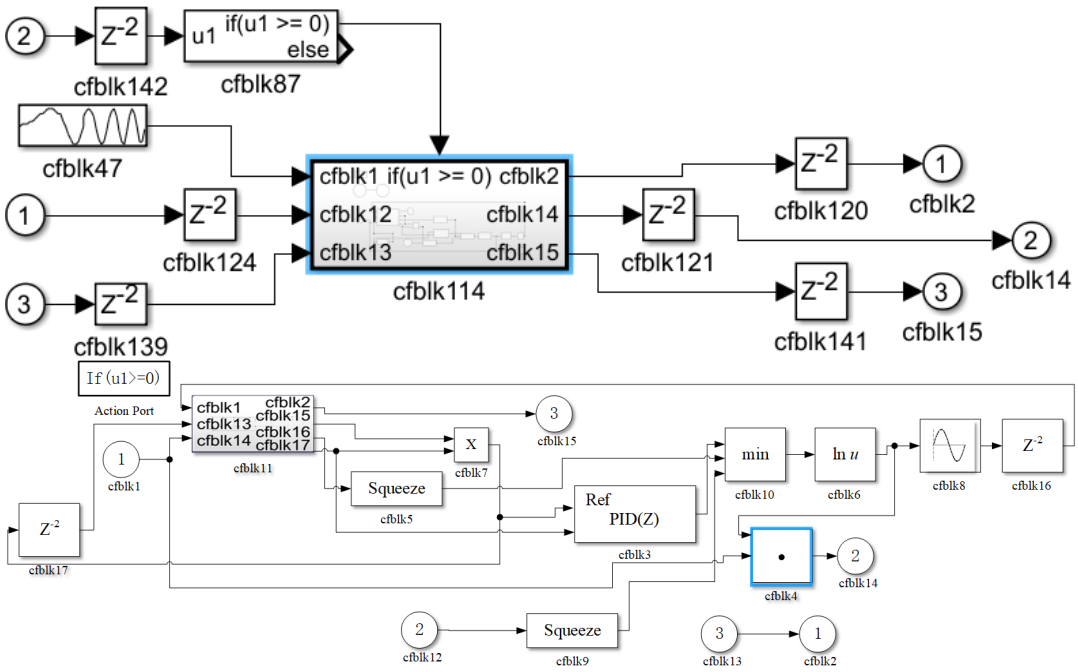

Fig. 6. TSC05246986: Data exceptions caused by different simulation operations

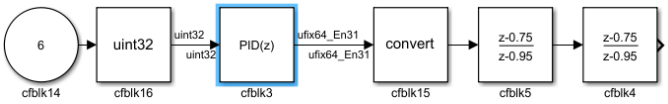

Fig. 7. TSC05372237: Unreasonable compilation error prompt

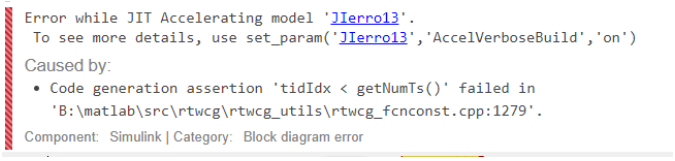

Fig. 8. TSC05385316: Model JIT accelerated simulation generate file failed

has no output data). Then, the seed CPS model and its variants are executed in the same Simulink setting to detect compiler bugs. In this RQ, we create four instances of SLEMI, including SLEMI(Default), SLEMI(Swarm), SLEMI(History), and SLEMI(RECORD). Each instance represents running SLEMI with CPS models generated by RECORD or one of the baselines. We run each SLEMI instance for two weeks due to the huge time cost. In each iteration, we first use RECORD or the baselines to generate 100 CPS models, which is the same setting as in RQ1. We then feed these CPS models into SLEMI for mutation. We use Simulink compiler to compile the seed CPS model and its variants in the same simulation mode. Specifically, we first compare the outputs of these CPS models in the *Normal* simulation mode. We then compile and compare them in the *Accelerator* simulation mode. If the outputs (i.e., output values or prompts) are different in any simulation mode for any two CPS models, a bug could be found. We run SLEMI with the source code provided in their study using default

TABLE V
NUMBER OF BUGS FOUND BY DIFFERENT APPROACHES + SLEMI

Approach	# of bugs
SLEMI(Default)	1
SLEMI(Swarm)	0
SLEMI(Hisotry)	2
SLEMI(RECORD)	4

parameters. For example, SLEMI generates 5 variants for each seed CPS model.

2) Result: As shown in Table V, SLEMI can find Simulink compiler bugs based on the CPS models generated by different CPS model generation approaches. However, CPS model generation approaches also influence the bug-finding capability of the EMI-based strategy. When we apply SLEMI on the CPS models generated by RECORD, four bugs are found in Simulink compiler. In contrast, SLEMI(Default), SLEMI(Swarm), and SLEMI(History) find 1, 0, and 2 bugs, respectively.

The reason is that SLEMI conducts mutation based on the structure of CPS models (e.g., zombie regions and subsystems). Compared with the other three approaches, RECORD can generate diverse CPS models with different structures. Although the number of CPS models generated by RECORD is smaller than the other approaches due to the time cost for configuration learning, the diversity of CPS models could potentially help SLEMI explore equivalent mutations in more Simulink compiler input space. As a result, more bugs are found by SLEMI(RECORD).

Answer to RQ2: *RECORD facilitates the EMI-based strategy to find more bugs compared to baselines.*

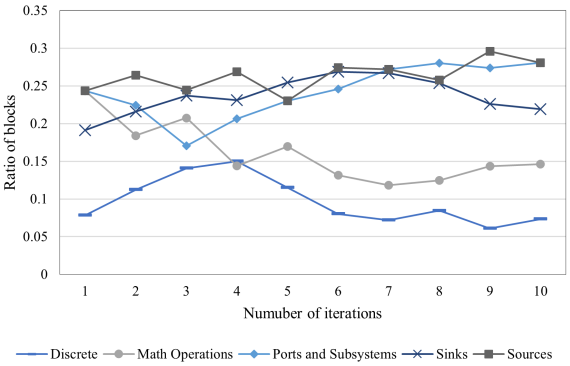

Fig. 9. Distribution of blocks in CPS models generated by RECORD

F. Impact of Configuration Learning (RQ3)

The reinforcement learning component in RECORD learns the relationship between configuration changes and rewards, which is then used to guide generating diverse CPS models. In this RQ, we analyze the distribution of blocks in different libraries generated by RECORD in each iteration to investigate the impact of the reinforcement learning component.

1) Methodology: We run RECORD to generate CPS models with different configurations for one episode. In each iteration, we collect a group of CPS models generated by RECORD, and compute the ratio of blocks in each library in these CPS models. Since SLforge only supports a subset of CPS model libraries [1], we mainly analyze the ratio of blocks related to *Discrete*, *Math Operations*, *Ports and Subsystems*, *Sinks*, and *Sources* libraries in this RQ.

2) Result: As shown in Fig. 1, RECORD attempts to generate CPS models with different block distributions. For example, initially the ratio of *Sinks* blocks is 0.19. During configuration learning, RECORD increases the probability to generate *Sinks* blocks to 0.27 to explore more input space of *Sinks* blocks. However, after a few iterations, RECORD finds the reward to add more *Sinks* blocks is decreasing. RECORD then configures the generator to reduce the probability to generate *Sinks* blocks. As a result, the ratio of *Sinks* blocks is down to around 0.22. We can observe similar increasing-decreasing (or opposite) trends for blocks in other libraries. For example, in this experiment, RECORD tends to generate CPS models with fewer *Ports and Subsystems* blocks at first. Since this may lead to simple CPS models with lower diversity, RECORD increases the ratio of *Ports and Subsystems* blocks to increase the complexity of CPS models. In this way, RECORD may have a higher probability to generate complex CPS models to trigger more Simulink compiler bugs.

According to the above observations, RECORD explores the input space of Simulink compiler by dynamically adjusting the distribution of different blocks. In this way, RECORD could efficiently explore the input space without generating too many "useless" CPS models, such as a CPS model with only a large number of *Sinks* blocks.

Answer to RQ3: *The reinforcement learning component improves the ability of RECORD to generate CPS models with diverse distributions of blocks.*

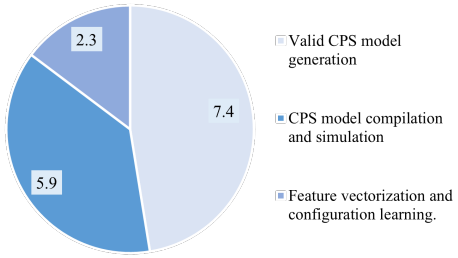

Fig. 10. Time distribution of executing RECORD (hours)

VI. Discussion

A. Efficiency

We spend a long evaluation period to test Simulink compiler. The time consumption of RECORD mainly consists of three parts, namely the time to generate valid CPS models, the time of CPS model compilation and simulation, and the time of feature vectorization and configuration learning. The pie chart in Fig. 10 shows the time distribution of each part. This time distribution is computed by running RECORD to generate 100 CPS models with different configurations. In the figure, valid CPS model generation takes the majority of time in the experiment, followed by the time of CPS model compilation and simulation.

The distribution of time in Fig. 10 can be understood as follows. First, we use SLforge to generate CPS models as inputs. The time to generate valid CPS models significantly increases as the complexity of CPS models. In our context, SLforge usually took 100–300 seconds to generate a CPS model with 1–5 hierarchy levels and 30–100 blocks. It may also reach the timeout threshold for generating some complex CPS models. Second, the execution of CPS models takes time. In the compilation phase, Simulink could take more than 10 seconds to compile a CPS model. In the simulation phase, since CPS models simulate the behaviors of the CPS, Simulink usually executes CPS models for a continuous period of time. For example, in Fig. 5 the input of the *cfblk60* block is a signal. We cannot execute this CPS model only once. The default configuration of this block is to run the CPS model for 10 seconds, and sample the signal every 0.1 seconds.

Hence, the bottleneck that affects the efficiency of Simulink testing is primarily the valid CPS model generation process. When the CPS model generator generates CPS models, it needs to search for available blocks and connections based on the semi-formal specification of CPS models. If the generation fails, the generator needs to fix the errors through multiple attempts. This issue is also discussed in the paper of SLforge [1].

Since RECORD can find bugs automatically without human intervention, the efficiency is not a main limitation of RECORD. To improve the testing efficiency, on the one hand, parallel computing can be used to speed up the testing process. For example, in the CPS model generation phase, we can generate multiple CPS models with different CPS model generator instances using a cluster of machines instead of two computers to find more bugs. The CPS model compilation and simulation can also be executed on a cluster of machines. On

the other hand, we can try simpler but efficient techniques for configuration learning (such as Default, Swarm, and History). However, the bug-finding capability of these techniques is not as effective as RECORD.

B. Choice of DDQN

We choose DDQN because, compared to DQN and other non-model methods of reinforcement learning, DDQN theoretically combines the advantages of these models, which addresses the problem of overestimation in the value estimation in DQN caused by data correlation. This advantage has been verified in existing studies [19]. In our preliminary analysis, besides DDQN, we tried typical reinforcement learning approaches with Matlab Reinforcement Learning Toolbox (e.g., Q-Learning). DDQN shows its better effectiveness. Hence, we use DDQN to adjust the configuration of CPS model generator to produce more diverse CPS models, thus facilitating the exploration of the bug space of Simulink compiler. Our experiment in Section V-F also demonstrates the effectiveness of DDQN.

As for the limitation of DDQN, DDQN is a typical learning-based approach, which requires data to iteratively learn the configurations to generate diverse and useful CPS models. Hence, at the first few rounds of iteration, DDQN usually randomly explores the configuration space due to the insufficient knowledge being learned, which leads to inefficient bug-triggering CPS model generation. However, as more feedbacks obtained by DDQN (i.e., the diversity of CPS models generated with different configurations), DDQN could better learn the strategy to generate diverse CPS models; hence more bug-triggering CPS models are generated.

C. Threats to validity

1) Internal threats: As a common threat for compiler testing, the effectiveness of RECORD depends on the functionality of CPS model generators (i.e., SLforge in this study). Since SLforge supports a subset of CPS modeling language specifications, RECORD can only diversify CPS models for blocks in certain libraries. However, blocks supported by SLforge are the most-used blocks in CPS models [1]. Using these language specifications, RECORD has identified several confirmed bugs in Simulink compiler.

In this study, RECORD outperforms baselines in terms of the bug-finding capability. However, as fuzzing testing approaches, the randomly generated configurations by swarm testing also have chances to find bugs detected by RECORD when it is executed for enough testing period. Since developers usually have limited time for software delivery, such approaches may not be feasible.

2) External threats: RECORD can trigger duplicate bugs, which could affect the comparison of different approaches. To alleviate this threat, we identify duplicate bugs by comparing their failed assertions and back-traces. Further, we submitted the detected bugs to developers for investigation. However, these duplicate bugs also mean that bugs found by RECORD are reproducible. RECORD can trigger a Simulink compiler bug with different CPS models.

Another threat is the generality of RECORD. In this study, we test the latest Simulink version. Similar to previous studies [3], [1], [2], we test Simulink because Simulink has become an industrial standard widely used for many safety-critical applications. We test the latest Simulink version, since developers confirm and fix bugs primarily in this version [22]. This version has been tested based on the latest test suite before release, which is more challenging to test. In our preliminary study, RECORD can also trigger bugs in other Simulink versions. However, developers suggest finding and de-duplicating bugs in new Simulink versions. To apply RECORD on other CPS tool chains or other types of compilers, the main changes are to redesign the features in Table II. In this study, we construct features based on the characteristics of CPS models (e.g., ports and subsystems, operations). To test other compilers, these features need to be adapted according to the language specification of the target compiler. However, the framework of reinforcement learning based configuration diversification proposed by RECORD is not limited by certain features.

VII. Related Work

The section discusses the related work on testing Simulink compiler and general-purpose compilers.

A. CPS Tool Chain Testing

A complementary line of work analyzes and looks for bugs in different components of CPS tool chains. Chowdhury et al. discuss the differential testing framework for testing arbitrary CPS tool chain [23]. Sampath et al. test CPS model-processing tools using semantic Stateflow meta-models [24]. Fehér et al. model the data-type inferencing logic of Simulink blocks for reasoning and experimental purposes [25]. Stürmer et al. test optimization rules of code generators utilizing graph grammars [26], [27].

Our work is related to testing the compilation system of CPS development tools. In this area, most of the works focus on Simulink compiler testing. Since CPS models are the basic inputs for Simulink compiler testing, Chowdhury et al. [3] propose CyFuzz for CPS model generation. CyFuzz randomly generates an initial CPS model and iteratively fixes its complication errors. To accelerate the generation process, SLforge [1] is proposed which generates CPS models based on the guidance of the specifications of the CPS modeling language. Since the formal specifications are neither complete nor publicly-available, DeepFuzzSL [28] and SLGPT [29] are proposed, which respectively use deep learning and transfer learning to automatically learn the relationship of block connections from existing CPS models to guide the generation. According to previous studies [1], [28], [29], SLfroge is the state-of-the-art CPS model generator in terms of the bug-finding capability and efficiency.

CPS models generated by CPS model generators can also be used as seeds for other testing strategies. SLEMI [2] is the first EMI-based Simulink compiler testing approach, which systematically mutates a seed CPS model as long as its semantics remain equivalent under a given input. The

seed CPS model and its equivalent variants are used to find Simulink compiler bugs.

Our work is different from these studies. On the one hand, we test Simulink compiler by guiding CPS model generators to generate diverse CPS models with different distributions of blocks. These CPS models can explore more input space of Simulink compiler to trigger bugs. On the other hand, RECORD facilitates the EMI-based strategy to find more bugs compared to other CPS model generation approaches.

Another line of work enables the CPS model testing for CPS tool chains. They mainly test CPS models fed into CPS tool chains. For example, MathWorks's Simulink Design Verifier [30] is an official tool from MathWorks, that uses static analysis to identify design errors in Simulink models. It can find CPS model errors such as array access violations, division by zero static, integer overflow, and static nested zombie blocks. DSVerifier [31] applies symbolic model checking to find design errors of CPS models. Nguyen et al. present a runtime verification framework for CPS model analysis [32]. These works test the correctness of CPS models, instead of CPS tool chain itself, which are different from our work.

B. General-purpose Compiler Testing

There are many works to test general-purpose compilers, such as GCC and LLVM. RECORD has a similar testing objective with swarm testing and history-guided testing [8], [9], which test GCC and LLVM by changing configurations of the C program generator Csmith. As discussed in Section II-D, there are two challenges to apply these approaches for Simulink compiler testing (i.e., the CPS model representation challenge and the configuration learning challenge). Experiments show that RECORD significantly outperforms these approaches in terms of the bug-finding capability on Simulink compiler.

Existing general-purpose compiler testing approaches can be classified into three categories, including, Randomized Differential Testing (RDT), Different Optimization Levels (DOL), and Equivalence Modulo Inputs (EMI) [33], [34], [35], [36], [37]. All these approaches are branches of differential testing. RDT detects compiler bugs by comparing the outputs of different compilers with the same specification. DOL compares the outputs produced by the same compiler with different optimization levels. EMI maintains the equivalent setting by generating equivalent program variants under a given test input [35], [36], [37]. All these approaches depend on random programs generated by program generators [33], [34]. Yang et al. [33] propose Csmith for C program generation, while Lidbury et al. [34] develop CLsmith to generate programs for testing OpenCL compilers.

Although the basic idea of RDT, DOL, and EMI could be applied for Simulink compiler testing, all these approaches require CPS models as inputs. RECORD can generate diverse CPS models to facilitate different testing strategies.

In addition to finding program optimization bugs, there are works to find bugs in certain components in general-purpose compilers, such as compiler warning bugs [38], [39], and link-time bugs [40]. Jiang et al. [22] proposed CTOS to test optimization sequence bugs in C compilers. RECORD is different from these works since we do not focus on testing these components in Simulink compiler.

VIII. Conclusion and Future Work

In this paper, we propose RECORD, an automated approach to generate diverse CPS models for Simulink compiler testing. RECORD includes a feature vectorization component and a reinforcement learning component, which address the CPS model representation challenge and the configuration learning challenge of Simulink compiler testing, respectively. RECORD can generate a large number of CPS models with diverse distributions of blocks in different libraries by intelligently altering the configurations of CPS model generators. RECORD significantly outperforms existing approaches for Simulink compiler testing. Within three months, we have reported 11 Simulink compiler bugs in total, of which 9 bugs have been confirmed as new by developers. As part of future work, we plan to test other CPS development tools and future versions of Simulink with RECORD.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No.62032004, No.62202079), the Dalian Excellent Young Project No.2022RY35, and the Fundamental Research Funds for the Central Universities under Grant DUT22ZD101, DUT22RC(3)028.

References

[1] S. A. Chowdhury, S. Mohian, S. Mehra, S. Gawsane, T. T. Johnson, and C. Csallner, "Automatically finding bugs in a commercial cyber-physical system development tool chain with slforge," in *Int'l Conf. on Software Eng. (ICSE)*. New York, NY, USA: ACM, 2018, pp. 981–992.

[2] S. A. Chowdhury, S. L. Shrestha, T. T. Johnson, and C. Csallner, "SLEMI: Equivalence modulo input (EMI) based mutation of CPS models for finding compiler bugs in simulink," in *Int'l Conf. on Software Eng. (ICSE)*. Washington, USA: IEEE, 2020, pp. 335–346.

[3] S. A. Chowdhury, T. T. Johnson, and C. Csallner, "Cyfuzz: A differential testing framework for cyber-physical systems development environments," in *Cyber Physical Systems Design, Modeling, and Evaluation*. Cham: Springer International Publishing, 2017, pp. 46–60.

[4] MathWorks Inc, "Products and services," http://www.mathworks.com/products/, 2018, (last access: 16/07/2022).

[5] J. Zander, I. Schieferdecker, and P. J. Mosterman, *Model-Based Testing for Embedded Systems*. Boca Raton, USA: CRC press, 2017.

[6] C. Guger, A. Schlogl, C. Neuper, D. Walterspacher, T. Strein, and G. Pfurtscheller, "Rapid prototyping of an eeg-based brain–computer interface (BCI)," *IEEE Trans. on Neural Systems and Rehabilitation Eng.*, vol. 9, no. 1, pp. 49–58, 2001.

[7] V. Pantelic, S. M. Postma, M. Lawford, M. Jaskolka, B. Mackenzie, A. Korobkine, M. Bender, J. Ong, G. Marks, and A. Wassyng, "Software engineering practices and Simulink: bridging the gap," *Journal on Software Tools for Technology Transfer*, vol. 20, no. 1, pp. 95–117, 2018.

[8] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program generation," in *Int'l Conf. on Automated Software Eng. (ASE)*. IEEE, 2019, pp. 305–316.

[9] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Int'l Symposium on Software Testing and Analysis (ISSTA)*, 2012, pp. 78–88.

[10] RECORD, "Record replication package," https://github.com/Simulink-Testing-Code/RECORD, 2022, (last access: 16/07/2022).

[11] MathWorks Inc, "Simulink documentation — matlab simulink," http://www.mathworks.com/help/simulink/, 2018, (last access: 16/07/2022).

[12] MathWorks Inc., "Simulink block libraries," https://www.mathworks.com/help/simulink/block-libraries.html, 2018, (last access: 16/07/2022).
[13] O. Bouissou and A. Chapoutot, "An operational semantics for simulink's simulation engine," *ACM SIGPLAN Notices*, vol. 47, no. 5, pp. 129–138, 2012.
[14] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using CLP," in *Int'l Conf. on Automated Software Eng. (ASE)*. Washington, USA: IEEE, 2015, pp. 482–493.
[15] G. Hamon and J. M. Rushby, "An operational semantics for stateflow," in *Int'l Conf. on Fundamental Approaches to Software Eng. (FASE)*. Berlin Heidelberg: Springer, 2004, pp. 229–243.
[16] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in GCC and LLVM," in *Int'l Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 294–305.
[17] MathWorks Inc, "Mathworks' bug reporting platform," https://www.mathworks.com/support/bugreports/, 2018, (last access: 16/07/2022).
[18] A. Plaat, *Deep Reinforcement Learning*. Springer, 2022. [Online]. Available: https://doi.org/10.1007/978-981-19-0638-1
[19] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," pp. 2094–2100, 2016. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12389
[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://doi.org/10.1038/nature14236
[21] MathWorks Inc, "Ddqn," https://ww2.mathworks.cn/help/reinforcement-learning/ug/dqn-agents.html/, 2022, (last access: 16/07/2022).
[22] H. Jiang, Z. Zhou, Z. Ren, J. Zhang, and X. Li, "CTOS: Compiler testing for optimization sequences of LLVM," *IEEE Trans. on Software Eng. (TSE)*, pp. 1–1, 2021.
[23] S. A. Chowdhury, "Understanding and improving cyber-physical system models and development tools," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 452–453.
[24] P. Sampath, A. Rajeev, S. Ramesh, and K. Shashidhar, "Testing model-processing tools for embedded systems," in *IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*. IEEE, 2007, pp. 203–214.
[25] P. Fehér, T. Mészáros, L. Lengyel, and P. J. Mosterman, "Data type propagation in simulink models with graph transformation," in *2013 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems*. IEEE, 2013, pp. 127–137.
[26] I. Sturmer and M. Conrad, "Test suite design for code generation tools," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 286–290.
[27] I. Stürmer, M. Conrad, H. Doerr, and P. Pepper, "Systematic testing of model-based code generators," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 622–634, 2007.
[28] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "Deepfuzzsl: Generating simulink models with deep learning to find bugs in the simulink toolchain," in *Workshop on Testing for Deep Learning and Deep Learning for Testing (DeepTest)*. New York, NY, USA: ACM, 2020, pp. 1–6.
[29] S. L. Shrestha and C. Csallner, "SLGPT: Using transfer learning to directly generate simulink model files and find bugs in the simulink toolchain," in *Evaluation and Assessment in Software Engineering (EASE)*. New York, NY, USA: ACM, 2021, pp. 260–265.
[30] MathWorks Inc, "Simulink design verifier matlab simulink," https://www.mathworks.com/products/sldesignverifier.html/, 2022, (last access: 16/12/2022).
[31] L. Chaves, I. Bessa, L. Cordeiro, D. Kroening, and E. Lima, "Verifying digital systems with matlab," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 388–391.
[32] L. V. Nguyen, C. Schilling, S. Bogomolov, and T. T. Johnson, "Runtime verification of model-based development environments," in *Proc. 15th International Conference on Runtime Verification (RV)*, 2015.
[33] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, 2011, pp. 283–294.
[34] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 65–76, 2015.
[35] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 216–226, 2014.
[36] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 386–399, 2015.
[37] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Int'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications (SIGPLAN)*. New York, NY, USA: ACM, 2016, pp. 849–863.
[38] Y. Tang, H. Jiang, Z. Zhou, X. Li, Z. Ren, and W. Kong, "Detecting compiler warning defects via diversity-guided program mutation," *IEEE Trans. on Software Eng. (TSE)*, pp. 1–1, 2021.
[39] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Int'l Conf. on Software Eng. (ICSE)*. IEEE, 2016, pp. 203–213.
[40] V. Le, C. Sun, and Z. Su, "Randomized stress-testing of link-time optimizers," in *Int'l Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 327–337.