Accelerating Reinforcement Learning-Based CCSL Specification Synthesis Using Curiosity-Driven Exploration

Ming Hu, Min Zhang, Frédéric Mallet, Xin Fu, Mingsong Chen

To cite this version:

Ming Hu, Min Zhang, Frédéric Mallet, Xin Fu, Mingsong Chen. Accelerating Reinforcement Learning-Based CCSL Specification Synthesis Using Curiosity-Driven Exploration. IEEE Transactions on Computers, 2023, 72 (5), pp.1431-1446. 10.1109/TC.2022.3197956 . hal-04178227

HAL Id: hal-04178227
https://inria.hal.science/hal-04178227v1

Submitted on 23 Oct 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Accelerating Reinforcement Learning-based CCSL Specification Synthesis Using Curiosity-Driven Exploration

Ming Hu, *Student Member, IEEE,* Min Zhang, *Member, IEEE,* Frédéric Mallet, *Member, IEEE,* Xin Fu, *Senior Member, IEEE,* and Mingsong Chen, *Senior Member, IEEE*

Abstract

The Clock Constraint Specification Language (CCSL) has been widely acknowledged as a promising system-level specification for the modeling and analysis of timing behaviors of real-time and embedded systems. However, along with the increasing complexity of modern systems coupled with strict time-to-market constraints, it becomes more and more difficult for requirement engineers to accurately figure out CCSL specifications from natural language-based requirement documents, since they lack both expertise in formal CCSL modeling and design automation tools to support quick and automatic generation of CCSL specifications. To solve the above problem, in this paper we introduce a novel and efficient Reinforcement Learning (RL)-based synthesis approach that can facilitate requirement engineers to quickly figure out their expected CCSL specifications. For a given incomplete CCSL specification, our approach adopts RL-based enumeration to explore all the feasible solutions to fill the holes within CCSL constraints, and leverages curiosity-driven exploration to accelerate the enumeration process. Based on the combination of our proposed curiosity-driven exploration heuristic and deductive reasoning techniques, our approach can not only prune unfruitful enumeration solutions effectively, but also optimize the enumeration process to search for the tightest solution quickly, thus the overall synthesis process can be accelerated dramatically. Comprehensive experimental results demonstrate that our approach significantly outperforms state-of-the-art methods in terms of both synthesis time and synthesis accuracy.

Index Terms

Clock Constraint Specification Language, Curiosity-Driven Exploration, Reinforcement Learning, Specification Synthesis

1 Introduction

Model-driven design and implementation of real-time and embedded systems have become an effective means of developing trustworthy systems [1], [2]. Modeling languages play an essential role in model-driven approaches. They are expected to formally and rigorously describe requirements, constraints, properties, and designs of the systems to be developed and meanwhile to be expressive and natural to engineers. The *Clock Constraint Specification Language* (CCSL) is such an emerging modeling language, companioning a UML profile called *Modeling and Analysis of Real-Time and Embedded systems* (MARTE) [3], [4]. CCSL has attracted attention from both industrial and academic communities [5], [6], [7]. A main feature of CCSL is that it devises *logical clocks* to model the temporal dependencies and constraints on system events. This feature abstracts physical details of events and focuses on only logical aspects, making CCSL more amenable to compose and more efficient to verify than physical clock-based languages. For these merits, CCSL has become a promising formalism at the *Formal Specification Level* [8] to accurately model and analyze the causal and temporal timing behaviors of real-time and embedded systems [9], [10].

Similar to other modeling languages, CCSL faces the same challenge when used to model real-world systems, i.e., going from natural language to formal models is an inherently difficult process as it means removing potential ambiguities from the natural language description. Picking one semantic variation over another one may have many unforeseen effects on the global behavior. The challenge has become one of the major obstacles in applying formal languages like CCSL to model complex real-time and embedded systems. Worse still, due to the skyrocketing complexity of real-time and embedded systems, it is almost impossible for requirement engineers to enumerate all the possible timing behaviors (traces) of a system, particularly at the early design phase. Consequently, CCSL specifications have to be composed mainly based on very limited expectations on system timing behaviors, stipulating what should happen (as specifications do). It makes the CCSL specification generation process both time-consuming and error-prone, requiring a number of refinement steps before the specifications are used for verification and validation purposes [11], [12]. Existing validation- and verification-based methods [13], [14], [15] and tools [16] for CCSL are proposed under the assumption that the specifications are complete and ready for verification. Although such methods are effective in detecting the flaws of specifications or verifying certain properties of them, it is a dilemma that incomplete specifications that are not “ready-to-verify” cannot be verified by any tool, presumably.

Specification synthesis has been recognized as an effective approach to generating formal specifications [17], [18]. It leverages automatic reasoning techniques to learn potential candidate solutions to the missing parts in incomplete specifications based on the collected system information such as execution sequences, logs, and behaviors. In the context of CCSL specification synthesis, most

- *Ming Hu, Min Zhang and Mingsong Chen are with the Shanghai Key Lab of Trustworthy Computing at East China Normal University, Shanghai, 200062, China (email: ecnu_hm@163.com, {zhangmin, mschen}@sei.ecnu.edu.cn). Frédéric Mallet is with Université Cote d'Azur, CNRS, Inria, I3S, France (email: Frederic.Mallet@inria.fr). Xin Fu is with the ECE Department at University of Houston, Houston, TX 77204, United States (email: xfu8@central.uh.edu).*

requirement engineers typically can figure out part of specifications based on their experience, leaving the unknown parts of causal and temporal constraints blank. As the first attempt to synthesize CCSL specifications, *CCSLSketch* [19], [20] enables the encoding of an incomplete CCSL specification together with its samples (i.e., expected timing traces) as a sketching problem, where the holes of incomplete CCSL constraints in the sketch can be filled automatically. However, since sketching is based on SAT solving, the scalability of *CCSLSketch* is strongly limited due to the *state space explosion* problem, especially for complex CCSL problems with a large number of correlated clocks and operators. Moreover, without considering the high-level syntax information of sketches, the underlying SAT solving of *CCSLSketch* may easily get stuck in local search, thus the synthesis time can be significantly prolonged. Therefore, *for a given incomplete CCSL specification, it is desired to devise an effective approach to quickly figure out an optimal hole filling scheme among all the hole combinations satisfying the given timing traces*.

Inspired by the curiosity mechanisms in [21], [22] and deduction work in [23], [24], this paper introduces a novel Reinforcement Learning (RL)-based CCSL specification synthesis method. To efficiently perform RL exploration, our approach tightly couples the merits of our proposed Curiosity-Driven Exploration (CDE) heuristic with deduction methods, thus accelerating the overall specification synthesis process. By formulating the syntax-guided CCSL specification synthesis based on hole enumerations as an RL problem, our approach formalizes a Markov Decision Process (MDP) for the RL training, where partial CCSL specifications are encoded as states and hole fillings are encoded as actions. Starting from an initial synthesis policy (i.e., a blank RL model) without any prior knowledge, our approach gradually improves the policy based on the MDP, which finally can be used to specify how to fill the holes of an incomplete CCSL specification to approximate its best-possible complete counterpart. To accelerate the convergence of policy generation, we resort to the help of our proposed CDE heuristic and deduction mechanisms as follows. Assume that *Spec* is an incomplete CCSL specification and *P* is the policy for *Spec*. By resorting to CDE, our approach can wisely fill one hole of *Spec* to get a more complete and suitable specification *Spec′*. Then, our approach uses the deductive reasoning methods to check whether *Spec′* is valid (i.e., *Spec′* admits the given traces) and update *P* accordingly based on our proposed reward mechanism. In this way, the deduction engine of our approach can refuse a filled CCSL specification at its earliest stage if it is invalid. The above RL training process repeats until *P* converges, which can finally be used to derive an optimal synthesis solution for the incomplete CCSL specification. Aiming at accelerating the above RL-based CCSL specification synthesis process, this paper makes the following **three major contributions**:

1) Based on our proposed state representation and reward design, we establish a novel RL-based synthesis framework, which involves various deduction reasoning mechanisms to enable efficient synthesis of CCSL specifications automatically.
2) We develop a new CDE method based on two tables (i.e., greed table and curiosity table) to support the wise selection of benign hole-filler candidates during the RL-based synthesis process, which can dramatically accelerate the search for optimal solutions.
3) We conduct experiments on both well-known benchmarks and complex industrial examples to show the effectiveness of our approach. Especially, our proposed CDE heuristic not only can reduce synthesis time significantly, but also further enhance the synthesis accuracy.

The remainder of this paper proceeds as follows. Section 2 presents the related work of program synthesis and CCSL specification formalization. Section 3 introduces CCSL notations. Section 4 describes the implementation of our curiosity- and deduction-driven approach in detail. Section 5 presents the results of performance evaluation. Finally, Section 6 concludes the paper.

2 Related Work

Along with the increasing popularity of CCSL in timing behavior modeling of real-time and embedded systems [5], [6], [7], [25], [26], more and more verification methods based on CCSL specifications have been studied to check various desired timing properties of target systems. For example, schedulability is one of the critical properties that should be formally guaranteed. If a specification is not schedulable, it implies that there must be a deadlock where no event can be triggered. To address this issue, Yin *et al.* [29] and Zhang *et al.* [30], [31] developed various effective techniques to support the schedulability analysis of complex CCSL specifications. Besides, more complex temporal properties formalized in temporal logic such as Linear Temporal Logic (LTL) are also verifiable [15] in CCSL. Apart from static analysis, runtime verification is another powerful approach in which the correctness of system behaviors is verified based on CCSL specifications at run time [25]. Although the above methods are promising in verifying the desired properties of CCSL specification from different perspectives, most of them assume that the CCSL specifications to be checked are all complete, which is not always true in practice. To ease the formal modeling process of CCSL specifications, a variety of approaches have been investigated, such as providing user-friendly GUI for requirement modeling engineers [16], transforming from other existing models [27], and iterative verification and refinement [28]. However, most of them require both human efforts and expertise. In this paper, our RL-based approach tries to reduce human intervention and efforts as must as possible in the modeling process.

As for specification synthesis, its effectiveness has been acknowledged to support the automated generation of formal specifications from requirements, system execution logs, or other textual descriptions [17], [18]. To support automated generation of CCSL specifications from incomplete templates, Hu *et al.* [19] proposed a promising method named CCSLSketch, which encodes the synthesis of an incomplete CCSL specification against provided timing behaviors into a sketching problem. However, since sketching is implemented based on Satisfiability Modulo Theories (SMT) solving that is essentially a generalization of the SAT problem, the applicability of CCSLSketch is inevitably limited. Generally, when dealing with complex CCSL specifications with multiple holes, CCSLSketch requires significantly more time and resources to achieve synthesis results. To reduce the time overhead, more and more specification synthesis methods resorted to RL techniques and deduction operations to explore desired enumeration solutions, since they are good at pruning unfruitful enumeration spaces. For example, $\lambda 2$ [32] and FlashMeta [33] adopted various effective methods, e.g., inverse

semantics and refutation, to decompose the synthesis task and guide the program search. By combining both deductive and statistical reasoning mechanisms, Chen *et al.* [23] introduced a domain-specific language synthesis algorithm, which uses RL to search for implementations under the guidance of an SMT-based deduction engine. In [24], Huang *et al.* proposed a cooperative synthesis framework called DryadSynth, which decomposes a synthesis problem into multiple subproblems and solves them individually based on both enumeration and deduction operations. Inspired by the above methods, to reduce the overall synthesis time, in this paper we propose a novel synthesis method for CCSL specifications based on RL-related techniques rather than using SAT- or SMT-based search.

As a promising heuristic to enable wise exploration on RL models, the curiosity mechanism has been widely investigated in RL to solve tasks with sparse rewards. Since curiosity-based methods enable more effective and sufficient search in RL, it has been used to potentially reduce the convergence time of RL training. For example, Pathak *et al.* [34] used the prediction error of forward dynamics models to some agent as an intrinsic reward to guide its exploration, where the extrinsic rewards to the agent are extremely sparse. In [21], Zheng *et al.* proposed an RL-based web testing framework that leverages a novel curiosity-driven reward function to efficiently explore diverse behaviors of web applications. In [22], Cao *et al.* introduced an effective human-machine interface software exploration framework named PathFinder, which uses a curiosity-based RL framework to choose actions that lead to the discovery of more unknown states. Although the above methods are promising in learning helpful knowledge for future scenarios, none of them can be directly applied on CCSL specification synthesis, since they are application-specific. To the best of our knowledge, our work is the first attempt that combines both deduction and CDE techniques to enable quick RL-based CCSL specification synthesis.

3 Preliminaries of CCSL

Logical clocks represent repetitive system events without assuming any regular distance between two successive occurrences. The "real-time" is treated as a special cases where successive tags associated with each occurrence are evenly spread. In CCSL, the logical clocks [35] are inspired from Leslie Lamport's logical clocks and clocks of the synchronous languages. Logical clocks give a most welcome elasticity when building specifications as the time is spread as much as needed to insert details when needed.

Definition 3.1 (Logical Clocks). A logical clock c is a predicate over natural numbers. For any $i \in \mathbb{N}^+$, $c(i)$ denotes that: i) if c is true at step i, c ticks at that step; or ii) if c is false, c is idle at step i.

According to their expressive power, logical clocks are classified into two categories, i.e., *atomic clocks* and *expression clocks*. An atomic clock is declared to represent a physical event in systems, while an expression clock is defined over existing clocks based on a certain relation. Such relations are defined by two types of operators, where *relation operators* (i.e., $O_b = \{=, \prec, \preceq, \subseteq, \#, \succ, \succeq, \supseteq\}$) are binary operators that define binary relations between clocks, and *expression operators* (i.e., $O_d = \{+, *, \wedge, \vee, \$, \propto\}$) are definitional operators that construct new clocks using other existing clocks.

TABLE 1
Syntax and Semantics of CCSL Operators

Constraint	Syntax	Semantics
Causality	$c_1 \preccurlyeq c_2$	$\chi_\delta(c_1,i) \geq \chi_\delta(c_2,i)$
Coincidence	$c_1 = c_2$	$\chi_\delta(c_1,i) = \chi_\delta(c_2,i)$
Exclusion	$c_1 \# c_2$	$c_1 \notin \delta(i) \vee c_2 \notin \delta(i)$
Precedence	$c_1 \prec c_2$	$(\chi_\delta(c_1,i) = \chi_\delta(c_2,i)) \implies c_2 \notin \delta(i)$
Subclock	$c_1 \subseteq c_2$	$c_1 \in \delta(i) \implies c_2 \in \delta(i)$
Delay	$c_1 \triangleq c_2 \$ d$	$\chi_\delta(c_1,i) = max(\chi_\delta(c_2,i) - d, 0)$
Infimum	$c_1 \triangleq c_2 \wedge c_3$	$\chi_\delta(c_1,i) = max(\chi_\delta(c_2,i), \chi_\delta(c_3,i))$
Intersection	$c_1 \triangleq c_2 * c_3$	$c_1 \in \delta(i) \iff c_2 \in \delta(i) \wedge c_3 \in \delta(i)$
Periodicity	$c_1 \triangleq c_2 \propto p$	$c_1 \in \delta(i) \iff c_2 \in \delta(i) \wedge \exists j \in \mathbb{N}^+ . \chi_\delta(c_2,i) = j \times p - 1$
Supremum	$c_1 \triangleq c_2 \vee c_3$	$\chi_\delta(c_1,i) = min(\chi_\delta(c_2,i), \chi_\delta(c_3,i))$
Union	$c_1 \triangleq c_2 + c_3$	$c_1 \in \delta(i) \iff c_2 \in \delta(i) \vee c_3 \in \delta(i)$

Table 1 presents both the syntax and semantics of all the operators involved in CCSL constraints defined as follows. Due to space limitation, please refer to [7] for more explanations about the formal semantics of CCSL operators.

Definition 3.2 (CCSL Constraints). Let c, c_1, c_2 be logical clocks. A CCSL constraint is in one of the following four forms:

$$c_1 \ ? \ c_2, \text{ where } ? \in O_b \tag{1}$$

$$c_e \triangleq c_1 \ ? \ c_2, \text{ where } ? \in O_d \setminus \{\$, \propto\} \tag{2}$$

$$c_e \triangleq c \ \$ \ d, \text{ where } d \in \mathbb{N}^+ \tag{3}$$

$$c_e \triangleq c \propto p, \text{ where } p \in \mathbb{N}^+ \tag{4}$$

In Table 1, the semantics of CCSL operators is formalized based on the notations of *schedule* and *history*. According to Definition 3.3, we use a schedule δ to describe a specific behavior of logical clocks. For a given time step i, $\delta(i)$ represents the set of clocks that tick at step i, where $\delta(i) \subseteq C$. In CCSL, schedules are partially reflected by system traces, where a *trace* denotes a finite sequence of system events logged by their occurring time. Typically, a CCSL specification consists of a collection of constraints to which all the feasible schedules should conform. We say that a CCSL specification admits a system trace if the trace conforms to all the specification constraints.

Definition 3.3 (Schedule). Given a clock set C, a schedule is a total function $\delta : \mathbb{N}^+ \to 2^C$ such that a clock $c \in C$ ticks at step $i \in \mathbb{N}^+$ if and only if $c \in \delta(i)$ and $\delta(i) \neq \emptyset$.

In CCSL, logical clocks need to record how many ticks they have ticked in the past up to the current step. Let χ_δ be the *history* of δ. For a given schedule δ and a clock c, we use $\chi_\delta(c,i)$ to denote the number of steps where c ticks before step i. In other words, $\chi_\delta(c,i)$ represents the number of ticks of c up to step $i-1$.

Definition 3.4 (History). Let δ be a schedule over clock set C. the history of δ is a function in the form of $\chi_\delta : C \times \mathbb{N}^+ \to \mathbb{N}$, where

$$\chi_\delta(c,i) = \begin{cases} \chi_\delta(c,i-1)+1 & \text{if } c \in \delta(i-1), \\ \chi_\delta(c,i-1) & \text{if } c \notin \delta(i-1), \\ 0 & \text{if } i = 1. \end{cases}$$

We take the semantic definition of precedence as an example to explain how CCSL constraints are defined by the concepts of schedule and history. Given a schedule δ and a constraint $c_1 \prec c_2$, we say that δ satisfies $c_1 \prec c_2$ (denoted by $\delta \models c_1 \prec c_2$) if and only if the condition $(\chi_\delta(c_1,i) = \chi_\delta(c_2,i)) \implies c_2 \notin \delta(i)$ holds for all $i \in \mathbb{N}^+$. Here, if c_1 and c_2 have the same number of ticks before step n, c_2 cannot occur at step n, i.e., c_1 always has more ticks than c_2. As a result, we can conclude that c_1 precedes c_2. That is, the tick of c_2 must be caused by a corresponding tick of c_1. Given a set of constraints, one fundamental issue known as *schedulability* [36] is to decide whether there exists a schedule satisfying all the constraints. If the set of constraints is not schedulable, there should be semantic conflicts among the constraints.

4 Our RL-Based CCSL Specification Synthesis Method

This section introduces a novel approach that can naturally convert the synthesis process of an incomplete CCSL specification into an RL problem. Unlike traditional RL approaches, our method fully investigates the synergy between both CDE heuristic and deductive mechanisms to guide the whole RL learning process, where the CDE heuristic is used to enable effective action selection and the deduction mechanisms are used to prune logically invalid candidate solutions. Under the guidance of the combination of CDE heuristic and deductive techniques, our approach can quickly achieve an optimal solution with significantly less effort compared with state-of-the-art methods.

4.1 Problem Formulation and Encoding

Due to the increasing complexity of real-time and embedded systems, it becomes more and more difficult to figure out complete CCSL specifications for timing behavior modeling at the beginning of requirement design. Typically, requirement engineers build CCSL specifications based on various kinds of modularized CCSL templates of system components in a bottom-up manner and leave various uncertainties as blanks (i.e., holes) during CCSL constraint design, resulting in incomplete specifications. An an example shown in [38], to model the timing behaviors of all the involved traffic lights of an intersection with a specific type (e.g., T-intersection, and cross intersection), requirement engineers can firstly construct an incomplete CCSL specification based on the CCSL template for a single traffic light, leaving the timing behaviors of traffic light interactions unspecified as holes. Based on the provided timing traces of expected traffic control scenarios, our approach can automatically and quickly figure out the CCSL specification for a specific-type interaction to enable accurate traffic control. In this subsection, we formalize the CCSL synthesis problem in terms of filling holes under the guidance of given system traces, and encode the CCSL synthesis as a classical RL problem based on a specific Markov decision process.

4.1.1 CCSL Specification Synthesis

An incomplete CCSL specification may contain four kinds of holes, i.e., $\square_o^b$, $\square_o^d$, $\square_c^b$, and $\square_c^d$, denoting holes for relation operators, expression operators, relation clocks, and expression clocks, respectively. Each hole indicates a placeholder for a specific clock or operator. For example, we use $\square_o^b$ and $\square_o^d$ to denote holes, whose candidate values come from O_b and O_d, respectively.

Definition 4.1 (Incomplete CCSL Constraint). An incomplete CCSL constraint is a constraint that contains one and only one hole h ($h \in \{\square_o^b, \square_o^d, \square_c^b, \square_c^d\}$) for either a clock or an operator.

An incomplete CCSL specification is a set including both regular constraints and incomplete constraints. The synthesis of incomplete CCSL specifications tries to automatically generate full CCSL specifications with all the holes filled according to the given auxiliary information. In our approach, we synthesize CCSL specifications based on the execution timing traces of systems, where each execution timing trace is a sequence of ordered events that occur temporally. It is required that a synthesized full (complete) CCSL specification should admit all the provided execution timing traces.

Definition 4.2 (CCSL Specification Synthesis Problem). For a given incomplete CCSL specification Φ and a set T of expected execution timing traces, the synthesis of Φ using T is to fill all the holes in Φ such that the completed specification can admit all the traces in T.

For a CCSL specification synthesis problem, based on a limited number of expected execution timing traces, there can be numerous solutions to filling holes. For two synthesized specifications s and s', we say that s is tighter than s' if s covers fewer traces that are not included in T than s'. In this paper, our synthesis method focuses on searching for the tightest solutions (i.e., *golden references*) to incomplete CCSL specifications. It is important to note that, due to semantic conflicts among the constraints, there may exist no solutions for filling the holes. In other words, the synthesis will fail eventually.

Definition 4.3 (Specification Tightness). Assume that Φ and Φ' are two synthesized results of an incomplete CCSL specification $\Phi_\Box$. We say that Φ is tighter than Φ' if and only if $\delta \models \Phi$ implies $\delta \models \Phi'$ for any schedule δ. If there exist no completed specification that is tighter than Φ, then Φ is the tightest solution to $\Phi_\Box$.

Since our approach is based on RL, the synthesis results may vary due to randomness. To enable fair comparison with state-of-the-art methods, we repeat the synthesis method for multiple times to achieve a set of synthesis results. We introduce the concept of *synthesis accuracy* as defined in Definition 4.4 to denote the ratio of the tightest solutions to all synthesis results in the set.

Definition 4.4 (Synthesis Accuracy). Let Φ be an incomplete CCSL specification and T is a set of its expected execution timing traces. Assume that $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ is a finite set of synthesized solutions under Φ and T, and $\Sigma' = \{\sigma'_1, \sigma'_2, \ldots, \sigma'_m\}$ is a subset of Σ containing all and only the tightest solutions. The synthesis accuracy equals to $\frac{m}{n}$, indicating the ratio of the tightest solutions in Σ.

4.1.2 MDP Encoding for CCSL Synthesis

Inspired by the success of RL in program synthesis [23], [37] and satisfiability problem solving [39], [40], the CCSL synthesis can be regarded as an RL process of filling candidate clocks or operators into the holes of incomplete constraints. This RL process can be modeled as an MDP [41], where agents get rewarded or punished based on their decisions on hole filling. In our approach, a state in the MDP is denoted by a list of holes, while an MDP transition indicates an action of filling one hole in the list. More precisely, the synthesis process of an incomplete CCSL specification can be formulated as an MDP in the form of a 4-tuple $M = (\mathcal{S}, \mathcal{A}, \mathcal{F}, \mathcal{R})$, where:

- $\mathcal{S}$ is a set of states, where a state is a list whose elements are in the form of a 2-tuple $\langle \Box^i, v \rangle$, indicating whether an indexed hole $\Box^i$ (i.e., a hole at index i) has been filled with some candidate value v. Here, we use the index i to specify the hole position in the list. If v equals *null*, it means that the hole $\Box^i$ has not been filled yet. At the very beginning of RL, none of the holes for an incomplete specification is filled. Therefore, for the initial state, the value of each element in the list is *null*. On the contrary, a terminal state is represented by a list with all the element holes filled, indicating a possible solution for the synthesis. To prevent RL from repeated searches, our approach requires that holes to be filled in a specific order. For example, assume that $\langle \Box^i, v_i \rangle$ and $\langle \Box^j, v_j \rangle$ are two elements in the same list (state). If $i < j$, $v_i = null$ implies $v_j = null$.
- $\mathcal{A}$ is a set of actions indicating filling holes with specific values. An element of $\mathcal{A}$ is a hole-value mapping. in the form of $(\Box^i, v)$, denoting an action that fills an indexed hole $\Box^i$ with some value v (i.e., a clock or an operator) depending on the hole type.
- $\mathcal{F} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is a transition function represented by a set of transitions, where each transition is in the form of $\alpha \xrightarrow{(\Box^i, v)} \alpha'$. Here α' is the successor state of α by assigning $\Box^i$ with a value v.
- $\mathcal{R} : \mathcal{S} \rightarrow \mathbb{R}$ denotes a reward function. By using a predefined function $\mathcal{R}(s) = RewardEvaluator(s)$, our approach adopts an evaluator to conduct a reward or punishment when RL reaches a state s.

Taking the CCSL specification shown at the bottom left of Figure 1 as an example, the incomplete specification has six CCSL constraints, where four of them are incomplete. It has six clocks including four atomic clocks (i.e., c_0, c_1, c_2, and c_3) and two expression clocks (i.e., e_0 and e_1). Based on the hole types, we can figure out the candidate values for each hole for the synthesis. For example, H_0 is an expression clock hole, and H_2 is a relation clock hole. Therefore, the values of H_0 and H_2 come from the clock set $C = \{c_0, c_1, c_2, c_3, e_0, e_1\}$. Since H_1 is an expression operator hole, its candidate value will be selected from the set $O_t = \{+, *, \wedge, \vee\} \subset O_d$. As a relation operator hole, the candidate value of H_3 belongs to the set $O_b = \{=, \prec, \preceq, \subseteq, \#, \succ, \succeq, \supseteq\}$. According to the definition, we construct one specific MDP $M = (\mathcal{S}, \mathcal{A}, \mathcal{F}, \mathcal{R})$ for the incomplete specification as follows.

- **State set** $\mathcal{S}$. In this example, an MDP state is a list in the form of $[\langle H_0, v_0 \rangle, \langle H_1, v_1 \rangle, \langle H_2, v_2 \rangle, \langle H_3, v_3 \rangle]$, indicating the filling of four holes. Initially, the values of all the holes in the list equal to *null*, i.e., $[\langle H_0, null \rangle, \langle H_1, null \rangle, \langle H_2, null \rangle, \langle H_3, null \rangle]$. After figuring out the hole types, we need to enumerate the holes one by one. The assignment of H_{i+1} is not allowed until H_i has been filled. We call a state a terminal state when all of its holes are filled, e.g., $[\langle H_0, c_0 \rangle, \langle H_1, * \rangle, \langle H_2, c_0 \rangle, \langle H_3, + \rangle]$. In total, the MDP has $|C| \times |C| \times |O_t| \times |O_b|$=1152 possible terminal states.
- **Action set** $\mathcal{A}$. For a hole H_i, we use $\mathcal{A}(H_i)$ to indicate a set of all possible actions that can be applied on H_i, where an element of $\mathcal{A}(H_i)$ is a mapping in the form of (H_i, v_i), indicating filling the hole H_i with a value v_i. Taking H_0 as an example, the action set of H_0 is $\mathcal{A}(H_0) = \{(H_0, c_0), (H_0, c_1), (H_0, c_2), (H_0, c_3), (H_0, e_0), (H_0, e_1)\}$. If no specific hole is specified, $\mathcal{A}$ denotes the action set for all the holes, which equals to the union $\mathcal{A}(H_0) \bigcup \mathcal{A}(H_1) \bigcup \mathcal{A}(H_2) \bigcup \mathcal{A}(H_3)$.
- **Transition set** $\mathcal{F}$. A transition $\alpha \xrightarrow{(H_i, v_i)} \alpha'$ in $\mathcal{F}$ indicates the effect of an action (H_i, v_i), where α and α' are a non-terminal state and its successor state, respectively. The transition is enabled only when either $i = 0$ or v_i is *null* and v_{i-1} is not *null*. If the transition is triggered, the value of v_i in the current state α will change from *null* to a candidate clock or operator.
- **Reward function** $\mathcal{R}$. Our reward function is the same as the one defined in [42], which can determine the quality (i.e., tightness) of specification synthesis results.

4.2 Overview of Our Approach

We adopt a classic RL algorithm, i.e., Q-learning [41], for CCSL specification synthesis. Figure 1 depicts an overview of our RL-based CCSL synthesis approach. The approach has two inputs, i.e., an incomplete CCSL specification (i.e., a set of incomplete CCSL constraints) and its corresponding system traces (i.e., execution timing traces). The output of our approach is a synthesized CCSL specification, which is complete and admits the given system traces. Our approach mainly consists of two steps, i.e., initial RL model generation by the model generator (see Section 4.1.2 for more details) and RL model training. At the beginning of specification

Fig. 1. Architecture and Workflow of our CCSL synthesizer accelerated by curiosity-driven exploration

synthesis, the model generator parses the two inputs and establishes a compact RL model including: i) a multi-layer network for MDP modeling and simulation, and ii) a Q-table for the training and generation of the best policy. As the key data structures of our approach, they are used in the RL training driven by our proposed CDE heuristic and deduction techniques. The RL training tries to explore the solution space of the given incomplete CCSL specification to quickly figure out an optimal solution. To make the exploration more efficient and sufficient, we use the CDE heuristic to guide the selection of candidate values. The deduction parts (i.e., CCSL checker and reward evaluator) have two main functions: i) checking whether an explored candidate solution admits the given system traces, and ii) performing reward evaluation for a candidate solution.

The workflow of our RL model training is as follows. After the establishment of an initial model based on input constraints and system traces, we use CDE to make the decisions of hole selection one by one until all the holes are filled. If one invalid hole filling is detected by the CCSL checker halfway, a penalty will be imposed on the Q-table. Otherwise, once all the holes are filled and the solution is valid, the reward evaluator will calculate the rewards for all the hole selection actions and update the Q-table accordingly. Note that, in our approach, one round of RL training involves only one update of the Q-table. In other words, any invalid hole selection made by the CCSL checker will lead to a new round of RL training (i.e., hole enumeration). Finally, the RL training terminates when either the training process converges or the specified number of training rounds ends. The following subsections present the key concepts and implementations of our approach in detail.

4.3 Initial RL Model Construction

Based on the MDP definition presented in Section 4.1.2, the goal of our Q-learning-based specification synthesis approach is to derive a policy π, which can reach a terminal state and get a maximum reward. To enable Q-learning-based synthesis, our model generator can automatically construct an initial RL model (i.e., a multi-layer network coupled with its Q-table) from an incomplete CCSL specification, where the underlying MDP procedure is simulated on top of the multi-layer network, and the Q-table is used to record credits for hole filling actions during the decision-making processes.

Based on the example presented in Figure 1, Figure 2 presents the major steps of initial RL model generation. Firstly, holes of the given incomplete specification are sorted and indexed according to their types, where each hole has multiple candidate values. Next, we construct a multi-layer network for MDP simulation, where each node in the network represents the result after filling some hole with a candidate value. A layer of the network indicates all the possible assignments of a specific hole. For instance, the hole H_0 has six candidate clocks, which can be used to complete the constraint $e_0 \triangleq \square_c^d + c_1$. As a result, the first network layer consists of six nodes. In the same way, the remaining layers contain four, six, and eight nodes, respectively. In the network, the layers are fully connected with each other, representing all possible solutions (i.e., hole enumerations) for the synthesis. To facilitate the modeling of solutions, we add two extra states, i.e., S and T, to represent the start and termination of one hole enumeration solution, respectively. A valid path from S to T indicates a feasible solution to filling all the specification holes.

In our RL-based approach, we use the reward mechanism to measure the quality of decision-making on hole filling, where the measuring results are reflected by a Q-table. The Q-table stores the expectation of the sum of rewards for each candidate, which can be used to guide the hole filling. Note that our approach does not adopt the traditional Q-table structure in Q-learning. This is because each row in a traditional Q-table indicates a state in the form of a list representing a path from S to a specific network node in the multi-layer network. Along with the increasing number of holes, the number of rows in the traditional Q-table will increases exponentially, which is not acceptable in practice due to extremely long RL-training time. Let us consider an example of the incomplete specification shown in Figure 2. If we use a traditional Q-table here, it requires $2+(6+6\times4+6\times4\times6+6\times4\times6\times8) = 1328$ rows, where “2” denotes

two states, i.e., S and T, and multiplication expressions in the parenthesis indicate the numbers of paths starting from S with different lengths (i.e., 1, 2, 3, and 4), respectively. As an alternative, in our approach all the nodes of a specific network layer share the same row within a Q-table, indicating the actions on a specific hole. For the example shown in Figure 2, our approach only needs four rows to construct a Q-table, thus the Q-table size can be dramatically reduced. Note that since different holes have different candidate values, the rows of our Q-tables may have different sizes. By default, all the Q-table cells are initialized to 0.

As shown in Figure 2, for the generated multi-layer network, each node (except S) of the multi-layer network connects to all the nodes of its predecessor layer. This is not necessary in practice, since there may exist semantic conflicts for some hole enumeration solutions. The full connections between network layers may easily result in unfruitful exploration in RL. For example, due to the conflict between the constraints $e_0 \triangleq c_3 + c_1$ and $c_3 \subseteq e_0$, it is impossible to fill hole H_0 with c_3. In other words, node $\langle H_0, c_3 \rangle$ can be safely pruned to accelerate the RL training, while the synthesis results are not affected. In our approach, we resort to the static pruning method based on the *rules of the number of clock ticks* proposed in [42] to remove such network constructs, thus achieving a compact initial RL model. For an obtained fully-connected multi-layer network, the pruning method tries to delete its unfruitful nodes based on all the given traces. By setting some Q-table cell with -1, we can disable the corresponding node in the multi-layer network, indicating that the hole enumeration path via this node is forbidden. Based on one trace with a length of 20, Figure 2 shows an example of model pruning, where the deleted nodes are greyed. Please refer to [42] for more details about the pruning of initial RL models.

4.4 RL Model Training

Algorithm 1 presents the implementation of our synthesis algorithm. Note that the input π_0 (i.e., Q-table) is a pruned initial model, where each valid cell of the Q-table is initialized with 0. Unlike classic Q-learning methods, our approach adopts two new data structures (i.e., *count table* T_c that records action selection statistics for each hole during RL, and *greed table* T_g that records the maximum reward explored so far for corresponding holes and actions). Please refer to Section 4.5.1 for more details about T_c and T_g. In line 3, we initialize both T_c and T_g using the Q-table, which are designed to guide the candidate selection in our CDE heuristic. Note that we use "$\pi_0 + 1$" to increase the values of all the valid Q-table cells by 1. The iteration at lines 5-21 details one RL round, which tries to select actions to fill all the holes of Φ. Line 10 denotes the action selection step, which we explain next in Section 4.5. Line 11 appends the selection result to *rList*, which forms the current state in terms of hole filling status. Line 12 updates the specification S by filling H with the selected action (H, a). Line 13 checks whether all the traces in T can be admitted by the updated specification, where the checking result is saved in r. Note that all the selections that cannot be determined by *CCSLChecker* so far are recorded in *unActList*. Please see Algorithm 3 for more implementation details about the function *CCSLChecker*. If r equals -1, line 15 will abort the current round and start a new round. Once all the holes are filled in one round, if r equals 1 and *unActList* is empty, the policy will be updated using *PolicyUpdate* (see Algorithm 4) in lines 18-20. Finally, based on the newly learned policy π_θ, line 22 returns a synthesized specification with all the holes filled.

Algorithm 1: Our CCSL Specification Synthesis Method

Input: **i)** Φ, incomplete specification; **ii)** T, set of execution timing traces; **iii)** π_0, pruned initial policy; **iv)** rd, # of RL training rounds.
Output: **i)** A synthesized specification for Φ.

```
1  RLSyn_DC(Φ,T,π0,rd) begin
2      πθ ← π0, maxSumRw← 0;
3      Tc ← π0+1, Tg ← π0;
4      for i ← 1 to rd do
5          S ← Φ;
6          rList ← [];
7          unActList ← [];
8          for j ← 1 to S.holes.len() do
9              H ← S.holes[j];
10             a ← ActionChoose(πθ,H,Tc,Tg,i,rd);
11             rList.append((H,a));
12             S ← Fill(S,(H,a));
13             (r,πθ,unActList,Tc)← CCSLChecker(S,rList,πθ,T,unActList,Tc);
14             if r == −1 then
15                 break;
16             end
17         end
18         if r == 1 ∧ unActList == [] then
19             (πθ,maxSumRw,Tc,Tg)← PolicyUpdate(S,rList,πθ,maxSumRw,Tc,Tg,i,rd);
20         end
21     end
22     return ResultEvaluator(Φ,πθ);
23 end
```

4.5 Curiosity-Driven Exploration

When dealing with a complex CCSL synthesis problem involving a large set of correlated holes (i.e., the holes that have dependencies on their fillings), existing RL methods may easily get stuck in the local enumerations, thus resulting in a long synthesis time. Moreover,

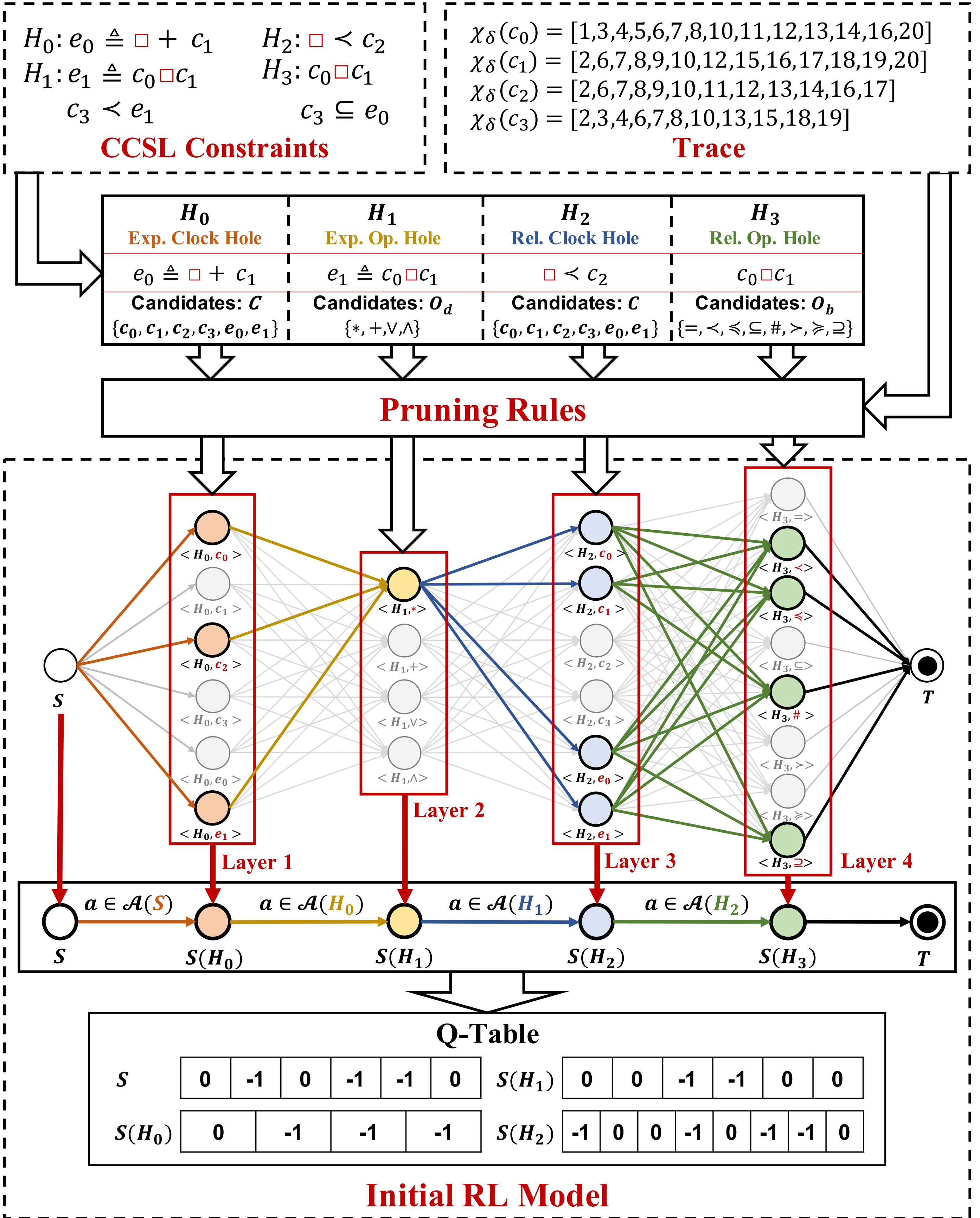


Fig. 2. An example of initial RL model generation with pruning

due to the inherent randomness of action selection, the insufficient search may strongly affect RL-based synthesis methods to quickly

converge to an optimal solution within limited rounds. For the example shown in Figure 2, assume that the RL only involves ten rounds in total for specification synthesis, where the first four rounds select hole candidates in a random manner and the last six rounds select hole candidates based on the Q-table with a probability 0.9 (i.e., $\epsilon = 0.1$). In this case, if a hole candidate is not touched in the first four rounds, its Q-value will be 0, where its chance to be selected in the last six rounds is very low, e.g., the probability of $\langle H_0, c_0 \rangle$ in the last six rounds is $(1 - [0.9 + (0.1 \times 2/3)]^6) = 14.1\%$. Let "$c_0$", "$*$", "$c_0$", and "$<$" be the optimal candidates for holes $H_0 - H_4$, respectively. The probability that $\langle H_0, c_0 \rangle$ is not selected in all the ten rounds is $((2/3)^4 \times [0.9 + (0.1 \times 2/3)]^6)$=16.12%. Similarly, the probabilities that $\langle H_2, c_0 \rangle$ and $\langle H_3, < \rangle$ are not selected in the last six rounds are the same, i.e., $((3/4)^4 \times [0.9 + (0.1 \times 3/4)]^6) =$ 27.18%. Therefore, the probability that the synthesizer cannot simultaneously cover all the four optimal candidates in all ten rounds is $(1 - (1 - 16.12\%) \times 100\% \times (1 - 27.18\%) \times (1 - 27.18\%))$=55.52%. In other words, the synthesizer has a 55.52% probability of getting stuck at local enumeration. Inspired by the curiosity-based RL method used in software testing [21], this paper introduces a novel CDE approach, which investigates the synergy between curiosity and greed to further enhance the efficiency and effectiveness of our RL-based synthesis method for action selection.

4.5.1 Design of Curiosity Mechanism

In our approach, we divide the whole RL training into two stages, i.e., *exploration-intensive stage* and *exploitation-intensive stage*. The exploration-intensive stage puts more focus on model exploration, where we use our proposed count table to ensure the quick coverage of all the model nodes. In other words, in this stage, we try to quickly establish a Q-table with acceptable performance. Based on the newly derived Q-table in the exploration-intensive stage and our proposed *greed table*, the exploitation-intensive stage focuses on wisely and efficiently searching for optimal solutions. Note that both the count table and the greed table have the same structure as the one of the Q-table.

In our approach, we maintain a count table (i.e., T_c) to record action selection statistics for each hole. For a hole H and an action a, we measure the curiosity by using Model-based Interval Estimation with Exploration Bonuses (MBIE-EB) [43]:

$$curiosity(H, a) = \frac{1}{\sqrt{T_c(H, a)}}, \quad (5)$$

where $T_c(H, a)$ denotes the corresponding count table cell value. Note that the value of $T_c(H, a)$ is initialized to 1. During the exploration, once a candidate action (H, a) is selected to fill the hole H, we increase $T_c(H, a)$ by 1. Equation (5) enables hole candidates that are selected with lower frequencies to have higher curiosity. During the exploration, the hole candidates with larger curiosity values will have higher chance to be selected firstly.

During the synthesis, some selected hole candidates may be independent of other holes. In this case, the hole candidates with the maximum rewards will be the ones in the final optimal solution. To facilitate the selection of such holes, we propose a greedy heuristic to assist the curiosity mechanism during the RL training. The heuristic uses a greed table (denoted by $T_g(H, a)$) to record the maximum reward explored so far for corresponding H and a, and adopts a rank function defined as follows to guide the candidate selection:

$$rank(H, a) = \begin{cases} curiosity(H, a) + \delta(i) \times T_g(H, a) & p \in [0, \eta) \\ curiosity(H, a) + \delta(i) \times Q(H, a) & p \in [\eta, 1] \end{cases}, \quad (6)$$

where p is a random real number variable whose value ranges between 0 and 1. Although there are two versions of rank calculation here, we can only choose one. In Equation (6), we use η to denote a specified probability. It means that the rank function has a probability η to use the first expression to calculate its value, and has a probability $1 - \eta$ to use the second expression to calculate its value. Note that in the exploration-intensive stage, we prefer a larger η, since it puts more weight for greed values. Alternatively, in the exploitation-intensive stage, we prefer a smaller η, since we need to mainly resort to the Q-table for selecting candidates. Let r be the given number of the overall training rounds. The notation $\delta(i)$ indicating a coefficient at the ith round is defined as $\frac{i}{r-i+1}$, whose value increases along with the increasing number of exploration rounds. Note that, at the beginning of the synthesis, both Q-table and greed table are not fully explored. In this case, we use δ (with a small value) to limit the weights of the greed table and Q-table, where the count table dominates the exploration. However, along with the increasing value of δ, the exploration becomes more inclined to select hole candidates using either greed table or Q-table rather than relying on the count table.

To wisely balance the effects of exploration and exploitation, our CDE heuristic uses the ϵ-greedy method to decide whether to use the rank function for hole candidate selection. Our approach chooses the candidate with the highest rank value with a probability of ϵ, and chooses the candidate based on the weight of each candidate with a probability of $1 - \epsilon$. We use the curiosity information to calculate the weight of each candidate that is defined as follows:

$$p(H, a) = \frac{curiosity(H, a)}{\sum_{a_i \in A(H)} curiosity(H, a_i)}, \quad (7)$$

where $A(H)$ is the candidate set of hole H. Generally, a candidate is more likely to be selected if it has a larger weight.

4.5.2 Curiosity-Driven Candidate Selection

Algorithm 2 details the implementation of our curiosity-driven candidate selection, where the inputs T_c and T_g are the count table and greed table, respectively. Line 2 obtains all the valid candidates of the hole H. Line 4 assigns p with a random value that is uniformly distributed in the range of $[0, 1]$. If $p > \epsilon$, lines 6-18 select a candidate with the maximum rank value. Lines 7-11 set the value of η to specify the stage of exploration, where $i/round < \beta$ indicates the exploration-intensive stage and $i/round \geq \beta$ denotes the exploitation-intensive stage. Line 13 calculates the rank value of the candidate, where the function *RANK* is defined based on Equation (6). If $p \leq \epsilon$,

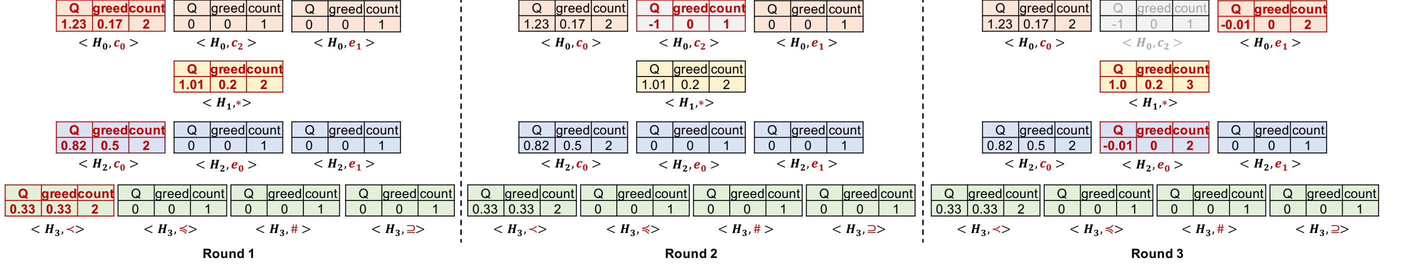


Fig. 3. An example of three consecutive rounds in our curiosity-driven exploration

Algorithm 2: Candidate Action Selection

Input: i) π_θ, policy; **ii)** H, hole to be filled; **iii)** T_c, count table; **iv)** T_g, greed table; **v)** i, i^{th} round of training; **vi)** rd, # of training times
Output: i) *candidate*, a selected candidate.

1 **ActionChoose**(π_θ,H,T_c,T_g,i,rd) **begin**
2 $\quad$ $candidateList \leftarrow \pi_\theta(H)$;
3 $\quad$ $candidate \leftarrow candidateList[0]$;
4 $\quad$ $p \leftarrow Random(0,1)$;
5 $\quad$ **if** $p > \epsilon$ **then**
6 $\quad\quad$ $mRank \leftarrow 0$;
7 $\quad\quad$ **if** $i/rd < \beta$ **then**
8 $\quad\quad\quad$ $\eta \leftarrow \eta_1$; // exploration-intensive stage
9 $\quad\quad$ **else**
10 $\quad\quad\quad$ $\eta \leftarrow \eta_2$; // exploitation-intensive stage
11 $\quad\quad$ **end**
12 $\quad\quad$ **for** *each c in candidateList* **do**
13 $\quad\quad\quad$ $r = RANK(H,c,T_c,T_g,\eta)$;
14 $\quad\quad\quad$ **if** $r > mRank$ **then**
15 $\quad\quad\quad\quad$ $mRank = r$;
16 $\quad\quad\quad\quad$ $candidate = c$;
17 $\quad\quad\quad$ **end**
18 $\quad\quad$ **end**
19 $\quad$ **else**
20 $\quad\quad$ choose $candidate \in \mathcal{A}(H)$ using *weight* info.;
21 $\quad$ **end**
22 $\quad$ **return** *candidate*;
23 **end**

line 20 selects a candidate based on its weight information (see Equation (7)) following the uniform distribution. Note that when all the holes are filled, we need to update both the count table and greed table (see line 8 in Algorithm 3 and lines 8-9 in Algorithm 4). Finally, line 22 reports the selected action.

Figure 3 presents an example to illustrate the candidate selection process during the exploration-intensive stage based on the pruned initial model shown in Figure 2. Here, each round involves four levels, where each level corresponds to one layer of the pruned model in Figure 2. For each node, we present its three values of the Q-table, greed table, and count table, respectively. Note that the cells of the Q-table, greed table, and count table are initialized to 0, 0, and 1, respectively. In each round, we need to compute a solution that is capable of filling all the holes. In the first round, since all the nodes in the same layer have the same count value, greed value, and Q-value, the selection of nodes can only be random. Assume that the nodes in the red color are the ones selected in the first round. Based on this selection, we need to update all the three tables (Q table by line 14 of Algorithm 4, greed table by line 9 of Algorithm 4, count table by line 8 of Algorithm 4). For example, the count values of the selected nodes are increased by 1. In the second round, starting from layer 1, since the count value of the three candidates $\langle H_0, c_0\rangle$, $\langle H_0, c_2\rangle$, and $\langle H_0, e_1\rangle$ are 2, 1 and 1, respectively, our approach tends to select $\langle H_0, c_2\rangle$ and $\langle H_0, e_1\rangle$ due to their smaller count values. We assume that $\langle H_0, c_2\rangle$ is selected here. Since $\langle H_0, c_2\rangle$ admits no given traces and this candidate is independent of other holes, we can prune this node from the model safely and terminate this round. Note that node $\langle H_0, c_2\rangle$ cannot be detected by the initial model pruning method proposed in [42]. In the third round, our approach selects $\langle H_0, e_1\rangle$, $\langle H_1, *\rangle$ and $\langle H_2, e_0\rangle$ as the candidates for the first three holes. However, such selection fails to admit the given trace, and the selection of $\langle H_2, e_0\rangle$ depends on the selection H_1. Therefore, we terminate this round halfway through and update the Q table (by line 7 of Algorithm 3) and count table (by line 8 of Algorithm 3) for the selected candidates in the round.

4.6 Deduction-Guided Checking and Reward Evaluation

To efficiently guide the hole filling and reward evaluation, we propose two deductive reasoning techniques: i) *deductive checking* that checks whether a filled constraint violates any provided traces; and ii) *deductive reward evaluation* that evaluates the reward of each hole filling action and updates the Q-Table correspondingly for a given complete specification.

4.6.1 Deductive Checking of Selected Actions

Algorithm 3: Deduction-guided Checking

Input: **i)** S, an incomplete specification; **ii)** *rList*, a list of actions already applied on holes; **iii)** π_θ, the policy; **iv)** T, a set of given execution timing traces; **v)** *unActList*, a list of uncertain actions; **vi)** T_c, count table.

Output: **i)** r, deductive checking result; **ii)** π_θ, an updated policy; **iii)** *unActList*, an updated list with unknown actions; **iv)** T_c, an updated count table.

```
1  CCSLChecker(S,rList,πθ,T,unActList,Tc) begin
2      (H,a)← rList.getLast();
3      r ← Deduce((H,a),T);
4      if r == −1 then
5          if checkDep(S,(H,a)) then
6              for (H′,a′)∈rList do
7                  πθ(H′,a′) ← πθ(H′,a′) − Γ;
8                  Tc[⟨H′,a′⟩] = Tc[⟨H′,a′⟩] + 1;
9              end
10         else
11             πθ(H,a) ← −1;
12         end
13     else if r == 0 then
14         unActList.append(H,a);
15     else
16         (r,πθ,unActList)← UpdateUnAct(unActList,πθ,S,T);
17     end
18     return (r,πθ,unActList,Tc);
19 end
```

Algorithm 3 presents the implementation details of our deductive checking method. Line 2 computes the latest hole filling action. Line 3 deductively checks whether the action can admit all the given traces, where the return value of *Deduce* is saved in r. Here, r has three values (i.e., -1, 0, and 1) indicating that the action is invalid, unknown, and valid, respectively. If the action is invalid (i.e, $r == -1$), lines 5-12 will update the policy. In line 5, we use the function *checkDep* to check whether the action (H, a) depends on other holes. Here, the dependence means that changing the assignments of other holes will affect the semantics of the current filled constraint. For example, assume that $e_0 \triangleq \square_0 + c_0$ and $e_0 \square_1 c_2$ are two incomplete constraints of a specification. Since the filling of hole $\square_0$ influences the behaviors of constraint $e_0 \square_1 c_2$, we say that $\square_1$ depends on $\square_0$. If the dependence exists, line 7 will update the policy by applying a small penalty Γ on the Q-values related to the actions stored in *rList*, and line 8 will increase the values of corresponding count table items by 1. Otherwise, line 11 will set the Q-value of the latest action (i.e., (H, a)) to -1, denoting that the node $< H, a >$ is removed from the multi-layer network. Note that, unlike the initial RL model pruning described in Section 4.3, line 11 denotes a new pruning technique based on the dependencies on hole assignments. Lines 13-14 indicate the case where the validity of the latest action is unknown. Taking the incomplete specification in Figure 2 as an example, assume that the latest action is filling H_0 with e_1. At this moment, the content of H_1 has not been determined yet, thus the timing behaviors of both e_0 and e_1 are uncertain. As a result, *Deduce* cannot determine the validity of action (H, a) at this stage. To record such uncertainties, line 14 appends the action to *unActList*. If the action is valid (i.e., $r == 1$), the function *UnpdateUnAct* in line 16 will check its impact on each unknown action in *unActList* and update *unActList* accordingly. If an unknown action in *unActList* becomes valid due to the latest action, it will be deleted from *unActList*. If an unknown action in *unActList* becomes invalid, *UpdateUnAct* will update both the policy and count table following the same way as shown in lines 5-12. Note that *UpdateUnAct* iterates until all the elements that can be inferred to be known in *unActList* are removed. Finally, line 18 reports the deductive checking results.

4.6.2 Deductive Reward Evaluation for Updating Policy

The reward mechanism plays an important role in determining the quality (i.e., tightness) of specification synthesis results. Based on the analysis of logical connections between CCSL constraints, we introduce a novel deductive reward evaluation method for given enumerated hole solutions, which can guide the policy generation towards the tightest synthesis results.

Algorithm 4 details our reward evaluation and policy updating process. Note that the algorithm assumes that input S is a complete specification with all the hole filling information saved in *rList*. For each action of hole filling, lines 3-10 calculate its rewards and update the count table and greed table accordingly. In each iteration, line 5 uses the function *rewardEvaluator* to evaluate the reward for the selection, where the reward is defined within the range $[0, 1]$. Note that we use different reward evaluation strategies for different hole types. Please refer to [42] for the detailed implementation of *rewardEvaluator*. Line 8 increases the corresponding items of count table T_c by 1. When the obtained reward is larger than the corresponding item in greed table T_g, line 9 needs to update the greed table. Lines 13-16 update the Q-table using the calculated rewards. To accelerate the convergence of Q-learning, the algorithm adopts the *dynamic learning rate* to enhance the reward and penalty effects. During RL, our approach is always inclined to learn from the search with the maximum sum of rewards. Meanwhile, our approach records the strategy with the highest reward sum explored so far. If the sum of rewards of a new search is greater than or equal to the record, we will increase the learning rate for the search. Otherwise, we will apply a lower learning rate on the search. Line 14 uses function $\alpha(\delta, i, rd)$ to calculate the dynamic learning rate. Note that the

Algorithm 4: Deduction-guided Reward Evaluation

Input: **i)** S, a complete CCSL specification; **ii)** *rList*, a list of actions applied on holes; **iii)** π_θ, the policy; **iv)** *maxSumRw*, maximum sum of rewards; **v)** T_c, the count table; **vi)** T_g, the greed table; **vii)** i, the ith round of training; **viii)** *rd*, total number of training rounds

Output: **i)** π_θ, an updated policy; **ii)** *maxSumRw*, an updated maximum sum of rewards; **iii)** T_c, an updated count table; **iv)** T_g, an updated greed table.

```
1  PolicyUpdate(S,rList,π_θ,maxSumRw,T_c,T_g,i,rd) begin
2      rewardList ← [], sumRw ← 0;
3      for j from rList.len() to 1 do
4          (H,a) ← rList[j];
5          reward ← rewardEvaluator((H,a),S);
6          rewardList.append(reward);
7          sumRw ← sumRw + reward;
8          T_c[⟨H,a⟩] = T_c[⟨H,a⟩] + 1;
9          T_g[⟨H,a⟩] = max(T_g[⟨H,a⟩], reward);
10     end
11     maxSumRw ← Max(maxSumRw, sumRw);
12     δ ← maxSumRw − sumRw;
13     for j from 1 to rList.len() do
14         π_θ(H,a) ← π_θ(H,a) + (sumRw − π_θ(H,a))×α(δ,i,rd);
15         sumRw ← sumRw − rewardList[j];
16     end
17     return (π_θ,maxSumRw,T_c,T_g);
18 end
```

values of α vary during the training. By default, if the reward sum obtained in the ith round is the largest (indicated by $\delta < 0$) so far, our approach will set α to: i) 0.8 when $i < rd/2$; or ii) 0.98 when $i \geq rd/2$. Otherwise, the α will be set to 0.1.

5 Performance Evaluation

To evaluate the effectiveness of our RL-based CCSL synthesis approach, we implemented our CCSL synthesizer using Python (version 2.7) and conducted experiments on various synthetic and industrial CCSL specification benchmarks. We compared our approach with the state-of-the-art method *CCSLSketch* [19], whose underlying sketching engine is based on SMT solving. Our experiments try to answer the following three research questions.

- **RQ1 (Superiority)**: What is the performance of our approach compared with the state-of-the-art method?
- **RQ2 (Scalability)**: What are the impacts of different settings (e.g., the number of rounds, trace lengths, the number of traces) on scalability?
- **RQ3 (Effects of CDE)**: Whether our proposed CDE heuristic can substantially improve synthesis time?

All the experimental results in this section were obtained from a laptop with 2.5GHz Intel i7 CPU and 16GB memory.

5.1 Experimental Settings

5.1.1 Benchmark Settings

We collected sixteen CCSL benchmarks (i.e., specifications) from four sources including: i) eleven benchmarks from the CCSL simulation tool *TimeSquare* [16] that investigates all the operators defined in Table 1; ii) three benchmarks (i.e., $Spec_1$-$Spec_3$) from *CCSLSketch* [19], [20], which is a state-of-the-art CCSL specification synthesis method based on sketching; iii) one complex synthetic benchmark (i.e., $Spec_4$) generated by ourselves manually; and iv) one complex benchmark (i.e., $Spec_5$) derived from an industrial land gear system [44] describing the timing behaviors of its aircraft undercarriage. To enable specification synthesis, firstly we used *TimeSquare* [16] to generate random timing traces with a specified length for each specification, which were used to guide the CCSL specification synthesis. Next, by randomly replacing clocks and operators from CCSL constraints with holes, we can form one incomplete version for each CCSL specification. According to Definition 4.1, each CCSL constraint within an incomplete specification has at most one hole for either a clock or an operator.

5.1.2 Parameter Settings

In the experiments, we set $\epsilon = 0.9$, and set $\beta = 30\%$ indicating that the exploration stage involves the first 30% RL rounds. We set $\eta = 0.5$ for the exploration-intensive stage, and set $\eta = 0.1$ for the exploitation-intensive stage. Since the given timing traces were derived from the collected CCSL specifications without holes, we treated them as the *golden reference specifications* for the synthesis. For a fair comparison, we applied our approach on each incomplete specification for 500 times. According to Definition 4.4, the synthesis accuracy in this case can be obtained with a precision of 0.2%. Note that requirement engineers can tune the number of repetitions to achieve their expected precision for the synthesis accuracy. Based on the obtained 500 synthesized specifications, we calculated the synthesis accuracy as the ratio of the synthesized specifications that have the same syntax as their golden reference specification. It is important to note that, for each incomplete CCSL specification, there may exist multiple synthesized specifications with different syntax formats satisfying the given timing traces. In this case, an accurately synthesized specification is the one that is as “tight” as its golden reference.

TABLE 2
Performance Comparison of Different CCSL Specification Synthesis Methods Using Collected Benchmarks

Source	Index	CCSL Statistics			Hole Settings			CCSLSketch	# of	RLSyn+D [42]		RLSyn+DC	
		Clk. #	Exp. #	Rel. #	Clk. #	Exp. Op. #	Rel. Op. #	Time (ms)	Rounds	Time (ms)	Acc. (%)	Time (ms)	Acc. (%)
TimeSquare [16]	$S_{=}$	2	0	1	0	0	1	867	10	4	50.8	4	100
	$S_{\prec}$	2	0	1	0	0	1	787	10	3	72.2	4	100
	$S_{\preceq}$	2	0	1	0	0	1	754	10	4	71.4	4	100
	$S_{\subseteq}$	2	0	1	0	0	1	791	10	3	100	4	100
	$S_{\#}$	2	0	1	0	0	1	685	10	4	100	4	100
	S_{*}	3	1	1	0	1	0	6639	10	8	100	8	100
	S_{+}	3	1	1	0	1	0	6787	10	7	100	7	100
	$S_{\vee}$	3	1	1	0	1	0	6922	10	7	100	7	100
	$S_{\wedge}$	3	1	1	0	1	0	6880	10	8	100	7	100
	$S_{\$}$	2	1	1	0	1	0	886	10	8	100	7	100
	$S_{\propto}$	2	1	1	0	1	0	908	10	6	100	6	100
CCSLSketch [20]	$Spec_1$	4	1	3	0	0	3	13782	10	6	60.4	6	100
									20	12	92.8	12	100
					0	1	2	36716	20	11	91.2	12	100
									50	29	98.4	30	100
	$Spec_2$	10	5	10	0	0	10	27689	50	69	81.8	70	98.6
									100	142	99.6	143	100
					4	2	4	40872	500	887	73.6	996	100
									1000	1807	96.8	1982	100
	$Spec_3$	20	6	16	9	1	7	433523	500	1506	57.8	1415	81.8
									1000	2956	92.8	2888	99.4
					5	1	10	42166	500	1412	87	1553	100
									1000	2898	96	3030	100
Synthetic	$Spec_4$	9	4	5	0	4	0	NA	50	101	61.2	105	94.6
									100	205	88.4	209	100
					5	1	1	NA	2000	3116	83.4	3378	99.4
									3000	4548	97.4	4997	100
LandGear [44]	$Spec_5$	25	74	54	1	1	5	NA	500	5091	75.8	5349	100
									1000	10325	85.8	10407	100
					3	1	0	NA	20	254	91.4	271	100
									50	805	99.8	706	100

5.2 Performance Comparison (RQ1)

This subsection highlights the benefits of our approach. We conducted a comprehensive comparison with the state-of-the-art method *CCSLSketch* using the collected benchmarks. To enable synthesis, for each benchmark we used *TimeSquare* to generate five traces with a length of 50 each. Table 2 presents the performance comparison results. The first column shows the sources of the collected specifications, and the second column presents the indices of the CCSL specifications. The third column consists of three sub-columns denoting the statistics of CCSL specifications, which includes the number of clocks, expressions, and relations for the collected CCSL specifications, respectively. The fourth column also has three sub-columns, which present the numbers of clock holes, expression operator holes, and relation operator holes, respectively. Due to space limitation, for each specification of $Spec_1$-$Spec_5$, here we provided only two different sets of incomplete constraints. The fifth column gives the synthesis time using *CCSLSketch*, where *"NA"* indicates the cases that result in the crash of *CCSLSketch* due to the notorious *state explosion problem*. The sixth column shows the maximum number of rounds used by the RL-based synthesis methods, where the first 30% rounds are for the exploration-intensive stage and the remaining 70% rounds are for the exploitation-intensive stage. To evaluate the performance (e.g., convergence) of different synthesis methods, for specific sets of CCSL constraints, we checked multiple scenarios with different numbers of rounds. The seventh column contains two sub-columns, which present the average synthesis time and accuracy information for our method, respectively, without considering the CDE. Note that the approach *RLSyn+D* is the same as the method proposed in [42]. Similarly, the eighth column shows the average synthesis time and accuracy information by taking both our deduction methods and CDE heuristic into account.

From Table 2, we can observe that compared with *CCSLSketch*, our RL-based methods can significantly improve the synthesis time by up to three orders of magnitude. Especially, we can find that *CCSLSketch* failed to synthesize all the incomplete specifications for both $Spec_4$ and $Spec_5$, while our RL-based approaches (i.e., *RLSyn+D* and *RLSyn+DC*) succeed in obtaining synthesis results. The reason why our approaches outperform *CCSLSketch* for $Spec_4$ and $Spec_5$ is mainly because our approaches are good at dealing with interrelated incomplete constraints, which cannot be efficiently handled by *CCSLSketch*. For example, "$? \prec e_2$, $e_2 = c_3 ? c_4$" are two interrelated incomplete constraints in the second case of $Spec_4$. Since the second case of $Spec_4$ has 5 clock holes, 1 expression operator hole and 1 relation operator hole, the size of its whole enumeration space is $(9+4)^5 \times 4 \times 8 = 11,881,376$. However, within 3000 rounds (enumerations), our *RLSyn+DC* method can achieve the tightest solution with an accuracy of 100%. It means that our method can achieve the same accuracy as *CCSLSketch* by exploring only 0.025% of its enumeration space, costing an average of 5 seconds for the CCSL specification synthesis. We can also observe that compared with *RLSyn+D*, although *RLSyn+DC* needs slightly higher time

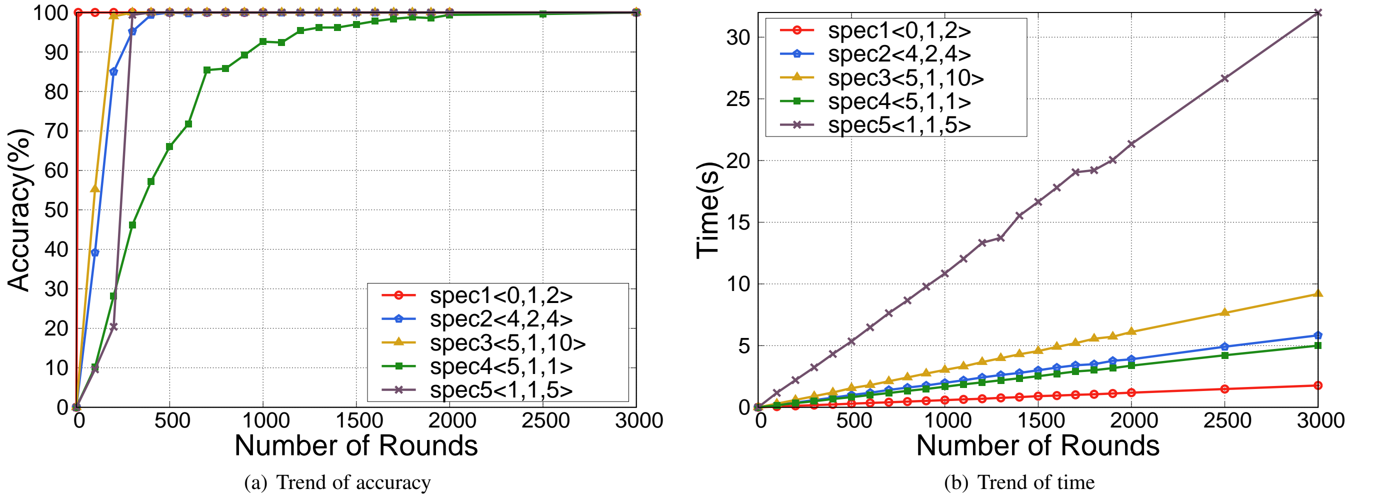
overhead under the same restriction of the number of rounds, it can achieve much better accuracy results. As an example for the first case of $Spec_3$, when there are only 500 rounds allowed, *RLSyn+D* has an accuracy of 57.8%, while *RLSyn+DC* can achieve an accuracy of 81.8%. Interestingly, we can find that in this case *RLSyn+DC* requires 5.8% less synthesis time than *RLSyn+D*, since *RLSyn+DC* converges earlier than *RLSyn+D*. Based on the above facts, in practice, we recommend using *RLSyn+DC* for synthesizing complex CCSL specifications.

5.3 Scalability Analysis (RQ2)

This subsection evaluates the impacts of three important factors (i.e., the number of RL rounds, trace lengths, the number of traces) on the scalability of our approach from the perspectives of synthesis time and accuracy. Note that all the synthesis results in this subsection were obtained by *RLSyn+DC*.

5.3.1 Impact of the Number of RL Rounds

Generally, the more rounds are involved in RL-based training, the more accurate synthesis results we can achieve. However, more rounds will inevitably lead to longer training time, which is not expected from requirement engineers. Therefore, it is better for RL-based synthesis methods to accurately derive the tightest specifications within a specific number of training rounds. To investigate the impacts of the number of rounds on specification synthesis, Figure 4 shows the accuracy and time trends for CCSL specification synthesis along with the increasing number of RL rounds. For conciseness, we only investigated one incomplete version indicated by the notation $Spec\langle x,y,z\rangle$ for each non-TimeSquare specification (i.e., $Spec_1$-$Spec_5$) shown in Table 2. Here, we use *Spec* to denote the specification index and x, y and z to indicate the number of holes for clocks, expression operators and relation operators within *Spec*, respectively. Note that the synthesis of each incomplete specification was guided by five random timing traces with a length of 50 each. When investigating the impact of a specific number of rounds on an incomplete specification, we need to reconduct the synthesis.



(a) Trend of accuracy

(b) Trend of time

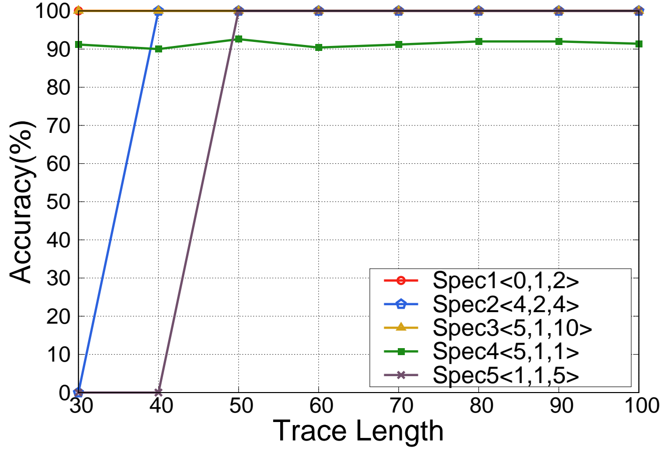
Fig. 4. Impacts of the number of rounds

As shown in Figure 4(a), when the number of RL rounds exceeds 2000, our method can obtain an accuracy of at least 99.4% for all the five specifications. When the number of rounds exceeds 3000, all the five cases achieve 100% accuracy. We found that the number of correlated holes is key to determining the synthesis accuracy by analyzing the hole types of these five incomplete specifications. When the number of rounds is fixed, the fewer correlated holes in some specification, the higher accuracy of the synthesis result we can achieve. For example, in this figure $Spec_4\langle 5,1,1\rangle$ has 4 correlated holes, while $Spec_3\langle 5,1,10\rangle$ has no correlated holes. In addition, from Figure 4(a) we can find that the number of rounds strongly affects the synthesis accuracy, where a larger number of RL rounds can always lead to a higher chance of finding the golden solution. This is because more RL rounds explore a larger proportion of the space of solutions. Note that in our approach one round involves one enumeration of all the holes, which costs almost the same time. Therefore, along with the increase of the number of rounds, the synthesis time shown in Figure 4(b) is increased linearly.

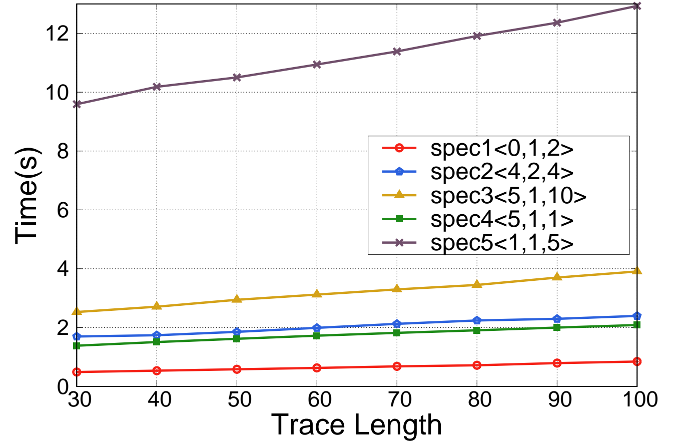
5.3.2 Impact of Trace Lengths

To illustrate the impacts of trace lengths, Figure 5 presents the synthesis results for the five incomplete specifications in terms of synthesis accuracy and time. In this experiment, we fixed the number of rounds to 1000. To enable fair comparison, at the beginning of each synthesis case, we randomly generated five timing traces with a length of 100. Then we used the prefixes with different lengths of these five timing traces to guide all the syntheses in the same case.

We can observe that, when the length of the five timing traces is 30, only three cases can achieve their highest accuracy. This is because, due to the limited behaviors of a target system reflected by the given timing traces with small lengths, it is hard for the evaluator to accurately figure out the rewards of multiple solutions. Consequently, there is a high chance that the synthesis results based on short timing traces may not be the golden ones. However, when the length of the five timing traces reaches 50, all of the five



(a) Trend of accuracy



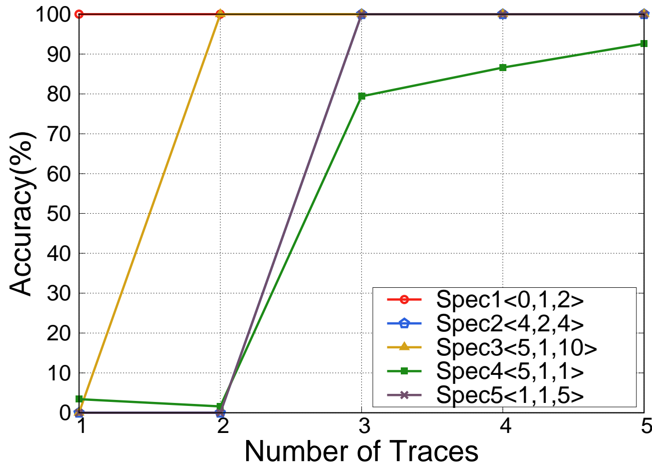
(b) Trend of time

Fig. 5. Impacts of trace lengths

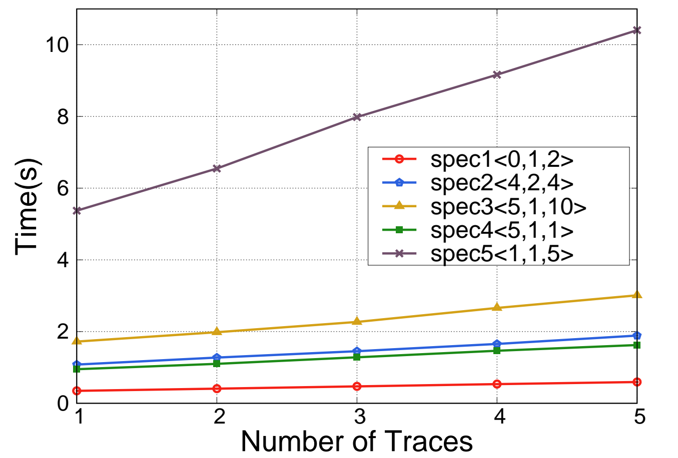
cases can achieve their highest accuracy. This observation means that the longer the timing traces are, which are given in input to the synthesis, the higher accuracy is achieved. Based on the same experimental settings used by Figure 5(a), Figure 5(b) shows the trend of synthesis time of the five cases. We can find that, along with the increase of timing trace lengths, the synthesis time of each case increases in a linear manner.

5.3.3 *Impact of the Number of Traces*

Figure 6 presents the synthesis results reflecting the impacts of the number of timing traces on both synthesis accuracy and time. In this experiment, the synthesis processes were guided by timing traces with a fixed length (i.e., 50), where each synthesis involves 1000 rounds. As shown in Figure 6(a), the more timing traces are offered in specification synthesis, the better synthesis accuracy our approach can achieve. In other words, longer (but presumably less) traces boost accuracy the same as more (but shorter) traces. This stands to reason, meaning that it is the total information value of the traces that counts. From this figure, we can find that when more than five timing traces are involved in the synthesis, *RLsyn+DC* can achieve almost the highest accuracy. Similar to the trends in Figure 4(b) and Figure 5(b), Figure 6(b) shows the linear increase trend of synthesis time.



(a) Trend of accuracy

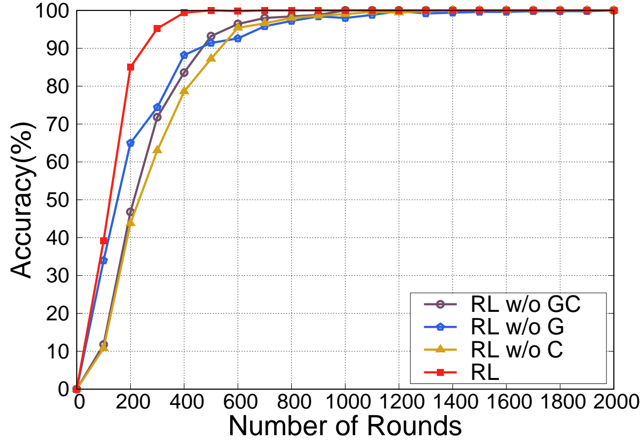


(b) Trend of time

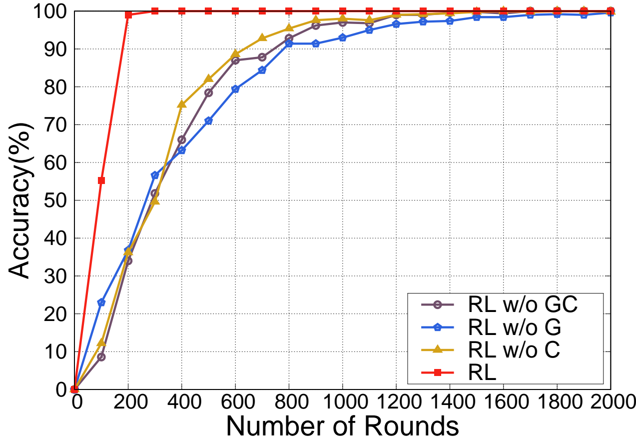
Fig. 6. Impacts of the number of provided traces

5.4 Ablation Studies for CDE (RQ3)

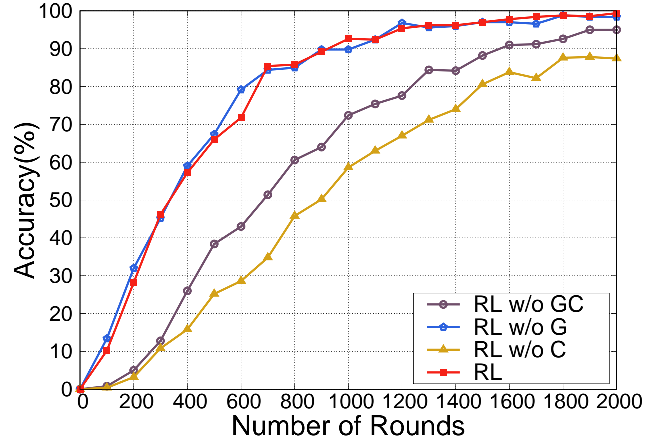
To evaluate the effectiveness of our proposed CDE techniques, we conducted ablation studies based on both the greed table and the count table. Note that, since our approach relies on deduction to evaluate the reward during RL and judge the correctness of hole filling operations, it is impossible to use the pure RL method for CCSL specification synthesis, thus we did not perform ablation studies for the deduction mechanisms. We designed three variants for *RLSyn+DC*, where the notation **RL** denotes the original version of *RLSyn+DC*. We use "**RL w/o GC**", "**RL w/o G**", and "**RL w/o C**" to denote the variant of *RLSyn+DC* without including both tables, the greed table and the count table, respectively.



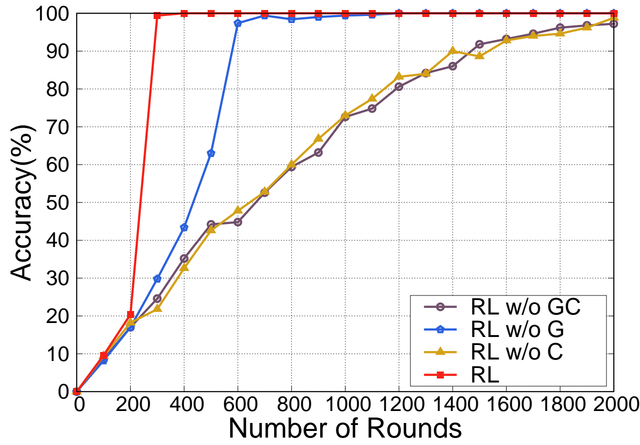
(a) Ablation study of $\mathrm{Spec}_2\langle 4,2,4\rangle$



(b) Ablation study of $\mathrm{Spec}_3\langle 5,1,10\rangle$



(c) Ablation study of $\mathrm{Spec}_4\langle 5,1,1\rangle$



(d) Ablation study of $\mathrm{Spec}_5\langle 1,1,5\rangle$

Fig. 7. Ablation studies for different specification benchmarks

Since $Spec_1\langle 0,1,2\rangle$ is too simple to truly reflect the effectiveness of our CDE techniques, we did not use it for the ablation study. Figure 7 shows ablation study results on the remaining four specifications used in the scalability analysis. From the four subfigures, we can find that **RL** can achieve the highest accuracy among the four synthesis methods, while the convergence requires the smallest number of rounds. As aforementioned in Section 4.5, the count table plays an important role in covering more candidates of hole combinations, thus it can be used to address the problem of stuck-at-local-search caused by correlated holes. We can observe that the count table has a more significant impact on accuracy improvement in Figure 7(c) and Figure 7(d). This is because $Spec_4\langle 5,1,1\rangle$ has four correlated holes, and $Spec_5\langle 1,1,5\rangle$ have two correlated holes, while the other two incomplete CCSL specifications have no correlated holes. By using the count table, hole values can be sufficiently enumerated during the exploration, which can in turn maximize the efficacy of the greed table. For the greed table, it is good at dealing with irrelevant holes when all of the hole values have been enumerated. For both Figure 7(a) and Figure 7(b), the impacts of either the count table or greed table on "**RL w/o GC**" individually are

negligible. However, based on their synergy, we can observe significant improvements in both the synthesis accuracy and convergence speed. Therefore, we suggest using both tables for the synthesis in practice.

6 Conclusion

Along with the increasing complexity and shortening time-to-market of real-time and embedded systems, Clock Constraint Specification Language (CCSL) is becoming more and more popular in their system-level design. However, since most requirement engineers lack the expertise in formal modeling of CCSL and corresponding efficient synthesis tools, it is hard for them to quickly and accurately derive expected CCSL specifications. To address this issue, we presented a novel approach that encodes the synthesis of incomplete CCSL specifications as a Reinforcement Learning (RL)-based hole filling problem based on our proposed state representations and reward mechanisms. Since the complexity of hole enumeration is exponential, we introduced a novel curiosity-driven exploration heuristic together with various deduction-based optimization techniques, which not only can effectively prune unfruitful enumeration space, but also accelerate the search for an optimal solution, thus the overall CCSL specification synthesis time is dramatically reduced. Experimental results on various benchmarks and industrial examples demonstrate the superiority of our approach in achieving higher synthesis accuracy and much shorter synthesis time.

Acknowledgments

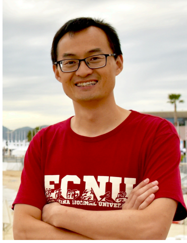
This work was supported by the National Key Research and Development Program of China 2018YFB2101300, Natural Science Foundation of China 61872147, and ECNU Academic Innovation Promotion Program for Excellent Doctoral Students YBNLTS2020-041. This work has been supported by the French government, through the France 2030 investment plan managed by the Agence Nationale de la Recherche, as part of the "UCA DS4H" project (ANR-17-EURE-0004). It has also been supported by the Inria associated team, Plot4IoT.

References

[1] G. Nicolescu, and P. Mosterman, "Model-based design for embedded systems,", CRC Press, 2018.
[2] M. Chen, X. Qin, H. Koo, and P. Mishra, "System-Level Validation: High-Level Modeling and Directed Test Generation Techniques,", Springer, 2013.
[3] Object Management Group, "UML profile for MARTE: Modeling and analysis of real-time embedded systems," 2011.
[4] J. Vidal, F. De Lamotte, G. Gogniat, P. Soulard, and J.P. Diguet, "A co-design approach for embedded system modeling and code generation with UML and MARTE," in *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2009, pp. 226–231.
[5] C. André and F. Mallet, "Specification and verification of time requirements with CCSL and Esterel," in *Proc. of ACM conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2009, pp. 167–176.
[6] J. Peters, R. Wille, N. Przigoda, U. Kühne, and R. Drechsler, "A generic representation of ccsl time constraints for UML/MARTE models," in *Proc. of Design Automation Conference (DAC)*, 2015, pp. 1–6.
[7] M. Zhang, F. Dai, and F. Mallet, "Periodic scheduling for marte/ccsl: Theory and practice," *Science of Computer Programming*, vol. 154, pp. 42–60, 2018.
[8] R. Drechsler, M. Soeken, and R. Wille, "Formal specification level: Towards verification-driven design based on natural language processing," in *Proceeding of Forum on Specification and Design Languages (FDL)*, 2012, pp. 53–58.
[9] F. Gao, F. Mallet, M. Zhang, and M. Chen, "Modeling and verifying uncertainty-aware timing behaviors using parametric logical time constraint," in *Proc. of Design, Automation and Test in Europe Conference (DATE)*, 2020, pp. 376-381.
[10] F. Mallet and M. Zhang, "Work-in-Progress: From logical time scheduling to real-time scheduling," in *Proc. of Real-Time Systems Symposium (RTSS)*, 2018, pp. 143–146.
[11] F. Thoen and F. Catthoor, "Modeling, verification and exploration of task-level concurrency in real-time embedded systems," Springer Science & Business Media, 2012.
[12] A. V. Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering," *IEEE Transactions on Software Engineering (TSE)*, vol. 26, no. 10, pp. 978–1005, 2000.
[13] J. Peters, N. Przigoda, R. Wille, and R. Drechsler, "Clocks vs. instants relations: Verifying CCSL time constraints in UML/MARTE models," in *Proc. of International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2016, pp. 78–84.
[14] F. Mallet and R. De Simone, "Correctness issues on MARTE/CCSL constraints," *Science of Computer Programming*, vol. 106, pp. 78–92, 2015.
[15] M. Zhang and Y. Ying, "Towards SMT-based LTL model checking of clock constraint specification language for real-time and embedded systems," in *Proc. of ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2017, pp. 61–70.
[16] J. DeAntoni and F. Mallet, "Timesquare: Treat your models with logical time," in *Proc. of International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, 2012, pp. 34–41.
[17] R. Drechsler, M. Soeken, and R. Wille, "Formal specification level: Towards verification-driven design based on natural language processing," in *Proc. of Forum on Specification and Design Languages*, 2012, pp. 53–58.
[18] F. Balarin and R. Passerone, "Specification, synthesis, and simulation of transactor processes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 26, no. 10, pp. 1749–1762, 2007.
[19] M. Hu, T. Wei, M. Zhang, F. Mallet, and M. Chen, "Sample-guided automated synthesis for CCSL specifications," in *Proc. of Design Automation Conference (DAC)*, 2019, pp. 1–6.
[20] CCSLSketch, https://github.com/HMHelloWorld/CCSLSketch.
[21] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic Web Testing Using Curiosity-Driven Reinforcement Learning" in *Proc. of IEEE/ACM International Conference on Software Engineering (ICSE)*, 2021, pp. 423–435.
[22] Y. Cao, Y. Zheng, S. Lin, Y. Liu, S. Yon, Y. Teo, Y. Toh, and V. Adiga, "Automatic HMI Structure Exploration Via Curiosity-Based Reinforcement Learning" in *Proc. of International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1151–1155.
[23] Y. Chen, C. Wang, O. Bastani, I. Dillig, and Y. Feng, "Program synthesis using deduction-guided reinforcement learning," in *Proc. of International Conference on Computer Aided Verification (CAV)*, 2020, pp. 587–610.
[24] K. Huang, X. Qiu, P. Shen, and Y. Wang, "Reconciling enumerative and deductive program synthesis," in *Proc. of ACM Conference on Programming Language Design and Implementation (PLDI)*, 2020, pp. 1159–1174.
[25] D. Yue, V. Joloboff, and F. Mallet, "Flexible runtime verification based on logical clock constraints," in *Proc. of Forum on Specification and Design Languages (FDL)*, 2016, pp. 1–8.
[26] X. Chen, Q. Liu, F. Mallet, Q. Li, S. Cai, and Z. Jin, "Formally Verifying Consistency of Sequence Diagrams for Safety Critical Systems," *Science of Computer Programming*, pp. 102777, 2022.

[27] X. Chen, L. Yin, Y. Yu, and Z. Jin, "Transforming timing requirements into CCSL constraints to verify cyber-physical systems," in *Proc. of International Conference on Formal Engineering Methods*, 2017, pp. 54–70.
[28] M. Montin, and M. Pantel, "Towards Multi-layered Temporal Models," in *Proc. of International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, 2021, pp. 120–137.
[29] L. Yin, J. Liu, Z. Ding, F. Mallet, and R. De Simone, "Schedulability analysis with CCSL specifications," in *Proc. of Asia-Pacific Software Engineering Conference (APSEC)*, 2013, pp. 414–421.
[30] Y. Zhang, F. Mallet, H. Zhu, and Y. Chen, "A logical approach for the schedulability analysis of CCSL," in *Proc. of International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2019, pp. 25–32.
[31] Y. Zhang, F. Mallet, H. Zhu, Y. Chen, B. Liu, and Z. Liu, "A clock-based dynamic logic for schedulability analysis of CCSL specifications," *Science of Computer Programming*, vol. 202, pp. 102546, 2021.
[32] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015, pp. 229–239.
[33] O. Polozov and S. Gulwani, "Flashmeta: A framework for inductive program synthesis," in *Proc. of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015, pp. 107–126.
[34] D. Pathak, P. Agrawal, A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction" in *Proc. of International Conference on Machine Learning (ICML)*, 2017, pp. 2778–2787.
[35] C. André, F. Mallet, and M.A. Peraldi-Frati, "A multiform approach to real-time system modeling: Application to an automotive system," in *Proc. of International Symposium on Industrial Embedded Systems (SIES)*, 2007, pp. 234–241.
[36] M. Zhang, F. Song and F. Mallet and X. Chen, "SMT-Based Bounded Schedulability Analysis of the Clock Constraint Specification Language," in the 22nd Fundamental Approaches to Software Engineering (FASE) 2019, pp. 61-78.
[37] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica, "AutoPandas: neural-backed generators for program synthesis," *Proc. of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2019, pp. 168:1–168:27.
[38] CCSL Synthesis For Traffic Signal Control, https://hmhelloworld. github.io/CCSLExampleTL/.
[39] V. Kurin, S. Godil, S. Whiteson, and B. Catanzaro, "Improving SAT solver heuristics with graph networks and reinforcement learning," *arXiv preprint arXiv:1909.11830*, 2019.
[40] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning sat solvers," in *Handbook of satisfiability*, 2021, pp. 133–182.
[41] M. Van Otterlo and M. Wiering, "Reinforcement learning and Markov decision processes," *Reinforcement Learning*, pp. 3–42, 2012.
[42] M. Hu, J. Ding, M. Zhang, F. Mallet, and M. Chen, "Enumeration and Deduction Driven Co-Synthesis of CCSL Specifications Using Reinforcement Learning," in *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 227–239.
[43] M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying Count-Based Exploration and Intrinsic Motivation," in *Proc. of Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 1471–1479.
[44] F. Boniol and V. Wiels, "The landing gear system case study," in *Proc. of International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ)*, 2014, pp. 1–18.
[45] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communication of ACM*, vol. 21, no. 7, pp. 558–565, 1978.

Ming Hu (S'19) received the BE degree from the School of Computer Science and Software Engineering, East China Normal University, Shanghai, China, in 2017. He is currently working toward the PhD degree in the Department of Embedded Software and System, East China Normal University, Shanghai, China. His research interests include the area of real-time system modeling and verification, program analysis, design automation of cyber-physical systems, and software testing.



Min Zhang (M'16) received the B.S. degree in computer science from Shandong Normal University, Jinan, China, in 2005, the M.S. degree in software theory from Shanghai Jiao Tong University, Shanghai, China, in 2008, and the Ph.D. degree in software science from Japan Advanced Institute of Science and Technology (JAIST), Nomi, Japan, in 2011. From 2011 to 2014, he was a Postdoctoral Researcher at JAIST. Afterward, he joined East China Normal University (ECNU), Shanghai, China, as a Professor. He is currently a Director of the Department of Software Science and Technology, ECNU. His research interests include formal methods, programming languages, and software engineering.

Frédéric Mallet (M'01) is a Professor of Computer Science in Université Côte d'Azur. He works on the definition of sound models and tools for the design and analysis of embedded systems and cyber-physical systems. He is a permanent member of the Kairos team, a joint team between Inria Sophia Antipolis research center and I3S Laboratory (CNRS). During several years, he has been a voting member of the OMG Revision Task Forces for MARTE and SysML, where he was leading the definition of the allocation subprofile and had a key role in the definition of MARTE Time Model and MARTE/CCSL. He has also contributed to the working group between MARTE RTF and AADL committee.

Xin Fu (M'09–SM'21) received the Ph.D. degree in Computer Engineering from the University of Florida, Gainesville, in 2009. She was an NSF Computing Innovation Fellow with the Computer Science Department, the University of Illinois at Urbana-Champaign, Urbana, from 2009 to 2010. From 2010 to 2014, she was an Assistant Professor at the Department of Electrical Engineering and Computer Science, the University of Kansas, Lawrence. Currently, she is an Associate Professor at the Electrical and Computer Engineering Department, the University of Houston, Houston. Her research interests include high-performance computing, machine learning, energy-efficient computing, mobile computing. Dr. Fu is a recipient of 2014 NSF Faculty Early CAREER Award, 2012 Kansas NSF EPSCoR First Award, and 2009 NSF Computing Innovation Fellow.

Mingsong Chen (M'08–SM'17) received the B.S. and M.E. degrees from Department of Computer Science and Technology, Nanjing University, Nanjing, China, in 2003 and 2006 respectively, and the Ph.D. degree in Computer Engineering from the University of Florida, Gainesville, in 2010. He is currently a Professor with the Software Engineering Institute at East China Normal University, and the director of Ministry of Education Engineering Research Center of Software/Hardware Co-design Technology and Application. His research interests are in the area of real-time computing, design automation of cyber-physical systems, EDA, parallel and distributed systems, and formal verification techniques. He is an Associate Editor of IET Computers & Digital Techniques, and Journal of Circuits, Systems and Computers.