Automated Synthesis of Safe Timing Behaviors for Requirements Models using CCSL

Ming Hu, Jun Xia, Min Zhang, Xiaohong Chen, Frédéric Mallet, Mingsong Chen

Automated Synthesis of Safe Timing Behaviors for Requirements Models using CCSL

Ming Hu, *Student Member, IEEE,* Jun Xia, *Student Member, IEEE,* Min Zhang, *Member, IEEE,* Xiaohong Chen, *Member, IEEE,* Frédéric Mallet, *Member, IEEE,* and Mingsong Chen*, *Senior Member, IEEE*

Abstract

As a promising requirement-level specification language for timing behavior modeling, the Clock Constraint Specification Language (CCSL) has become popular in the model-driven design community for safety-critical embedded systems. However, due to the skyrocketing design complexity, in practice it is hard for requirement engineers to accurately construct requirement models with expected timing behaviors using CCSL, especially for safe timing behaviors. Although more and more CCSL synthesis approaches are designed to facilitate the generation of CCSL specifications, most of them cannot be used directly for the synthesis of requirements models. This is because existing CCSL synthesis methods: i) focus on filling the holes of CCSL constraints rather than completing requirements models; and ii) rely heavily on limited observations of system behaviors, while the (temporal) safety properties of target systems are neglected. To address these issues, this paper proposes a novel method that enables the automated synthesis of safe timing behaviors for requirements models. By specifying the safety timing properties of target systems using safely-LTL, our approach adopts CCSL as an intermediate representation of requirement synthesis, where incomplete requirements models coupled with safely-LTL-based properties are encoded into CCSL constraints with holes. Guided by the samples (expected behaviors) provided by requirement engineers, our approach can automatically figure out the complete version of incomplete requirements models. Comprehensive experimental results on two complex case studies demonstrate that our approach can not only quickly and efficiently synthesize requirement models, but also guarantee that the synthesized models satisfy specified safety properties in Safely-LTL form.

Index Terms

CCSL, Synthesis, LTL, Requirement Model, Safety Timing Property.

I. Introduction

Due to steadily increasing complexity coupled with time-to-market pressure, it becomes more and more challenging to design safety-critical real-time embedded systems (e.g., automotive, avionics, robotics, and railway control) quickly and correctly [1], [2], [3], [4]. To address this issue, modern embedded system design adopts well-known *model-based requirement engineering methods* [5], [6] striving for higher-level requirements models, which help software engineers to accurately figure out the desired behaviors (especially timing behaviors [7], [8]) of target systems at an early design stage. By resorting to formal/semi-formal specifications such as UML (Unified Modeling Language) and its variants (e.g., MARTE, SysML), requirement engineers can construct various behavioral requirements models in an unambiguous manner [9], [5]. As initial models in the top-down design flow, the derived requirements models can not only be used to perform early validation and verification to detect crucial design flaws [10], but also provide a bridge between requirements management tools and system models [11], [2].

So far, most of the initial requirements are specified in natural language texts, which are then transformed into requirements models [12], [13], [14]. This process could be time-consuming and error-prone, since most requirement engineers have limited domain knowledge and formal modeling expertise in elaborating such models. Worse still, due to the increasing interactions with the external environment, the complexity of timing behavior modeling and analysis for embedded systems is skyrocketing. As a result, most requirement models are constructed simply based on limited observations and expectations of system timing behaviors. In this situation, it is hard for requirement engineers to derive a comprehensive requirement model satisfying all the given safety requirements, specifying what the system should not do [15]. This further makes the generated requirement models more error-prone. A major bottleneck is *a lack of design methodologies and supporting tools to enable the automated generation of requirements models for requirement engineers with limited knowledge in formal modeling of timing behaviors.*

As a promising specification language for requirements modeling, MARTE [16], [17], a UML profile for *Modeling and Analysis of Real-Time and Embedded systems* has put more and more emphasis on the modeling of timing behaviors for systems. MARTE adopts a companion formal specification language named *Clock Constraint Specification Language* (CCSL) [18], [19], [20] to model the timing behaviors of real-time embedded systems accurately. Based on the concept of *logical time* introduced by Leslie Lamport, CCSL offers a comprehensive set of predefined operators (i.e., clock operator and relation operator) to capture classical causal and temporal relations among events, where logical clocks act as first-class citizens and model/control/observe repetitive system events. Although CCSL is good at the modeling of timing behaviors of systems and there exists a wide spectrum of validation- and verification-based methods (e.g., through automata [21], transition systems

The authors Ming Hu, Jun Xia, Min Zhang, Xiaohong Chen and Mingsong Chen are with Shanghai Key Lab of Trustworthy Computing at East China Normal University, Shanghai, 200062, China. The author Frédéric Mallet is with Université Cote d'Azur, CNRS, Inria, I3S, France.
*Corresponding author. Tel: +(8621) 62235116; fax: +(8621) 62235255. E-mail address: mschen@sei.ecnu.edu.cn

[22], [23], and SMT solving [24]) and tools (e.g., TimeSquare [25]) that support the error detection and model checking of CCSL specifications, there remains a large gap between informal requirements and a very precise timing description, difficult to fill unless there is some assistance from specification generation tools for CCSL [26]. This is because even a simple timing behavior requirement may involve a large set of CCSL constraints. Unfortunately, since few existing methods support the automated transformation from requirements models to CCSL constraints, it is hard for a requirement engineer with common knowledge of formal methods to accurately specify the expected timing behaviors of target systems.

To alleviate this problem, various CCSL synthesis methods have been proposed to facilitate CCSL specification starting from scratch or incomplete templates. Generally, existing CCSL synthesis methods can be classified into two categories, i.e., constraint solving-based methods [27], and reinforcement learning-based methods [26]. However, both kinds of synthesis methods are based on the assumption that only sample behaviors (i.e., timing traces) of target systems can be obtained during the requirements modeling stage. It is hard for these approaches to guarantee the safety properties of timing behaviors of synthesized CCSL specifications, which are essential for safety-critical systems. Moreover, CCSL specifications are not suitable to be directly used for constructing requirements models. This is because a typical CCSL specification involves numerous logical clock-based timing constraints, where each constraint only indicates the relation between two events. Compared with traditional model-based design for requirements, such constraints are too counter-intuitive to describe the overall system behaviors, since they cannot describe functional requirements (e.g., control flow and data flow) of systems. In other words, the CCSL constraints can only be considered as the implementation-level formalism to detail the timing relations between system events.

Inspired by the concept of synthesis-based engineering [28], in this paper we propose a novel and effective timing behavior synthesis method for requirements models, where the system functional behavior model part and safe timing behavior requirements can be modeled separately. Given an incomplete requirements model S, a set of expected timing behaviors τ of the target system, and a set of safe timing properties P in the form of safely-LTL formulas, our synthesis method can figure out a complete version of S (i.e., S') by the incremental elaboration of timing behaviors of S in a correct-by-construction manner such that $S' \models \tau \wedge P$, indicating that both τ and P are all satisfied by S' in every situation. Note that in our approach, requirements engineers only need to focus on "*what should the requirements models do*" (i.e., the construction of S) rather than on "*how must the requirements models achieve P*". Unlike formal verification-based methods, our synthesis approach can produce a complete requirements model automatically satisfying both τ and P in a single iteration. Therefore, verification is no longer needed as S' is correct-by-construction. This paper makes **following three major contributions**:

- We introduce a novel timing behavior synthesis framework by using CCSL as the underlying Intermediate Representation (IR), where both functional behavior models and timing behavior requirements can be uniformly encoded into CCSL constraints.
- We propose an efficient safely-LTL formulas to CCSL constraints encoding method that can not only apply safety requirements of timing behaviors on requirements model synthesis but also generate concise CCSL constraints to accelerate the synthesis process.
- We improve the state-of-the-art CCSL synthesis engines to support flexible CCSL synthesis under the guidance of both expected timing behaviors and counterexamples.

Comprehensive experimental results on two non-trivial case studies (i.e., traffic signal control and high-speed train control) show the effectiveness and efficiency of our proposed method in automatically deriving requirements models.

The rest of this paper is organized as follows. Section II introduces the related work of requirements model synthesis and CCSL-based synthesis. Section III presents the preliminaries and notations used in the CCSL-based requirements model synthesis. Section IV details our automated timing behavior synthesis method for requirements models using CCSL. Section V presents the experimental results. Finally, Section VI concludes the paper.

II. Related Work

Due to the merits of improved design quality and reduced efforts and costs, the concept of synthesis-based engineering [28] has gained more and more attention in different activities (e.g., designing, programming, and testing) and components (e.g., environment, specifications, programs) of the software engineering development process, especially the embedded control systems [29], [30]. For example, Letier and Heaven [31] introduced deontic input output automata and present a formal framework for automated specification synthesis that is suitable in a requirements engineering context. Prabhu et al. [32] proposed a method for finding maximal and non-vacuous specifications, where the maximality allows for more choices of undefined procedure implementations and non-vacuity ensures the reachability of safety assertions. Li et al. [33] introduced the first framework named FlashRegex that can generate anti-ReDoS regexes by either synthesizing or repairing from given examples. Wang et al. [34] developed a novel method ARepair, which combines both mutation testing and program synthesis to provide an effective solution for repairing Alloy models. Based on efficient model checking and SMT solving, Ceska et al. [35] presented a counterexample-guided inductive synthesis (CEGIS) method to automatically synthesize finite-state probabilistic programs. In [36], Finkbeiner et al. developed the requirement synthesis tool RESY, which can automatically compute environment assumptions for compositional model checking. Gao et al. [37] developed a SKETCH-based compiler named Chipmunk, which formulates switch machine code generation as a program synthesis problem. Obergfell et al. [38]

investigated the synergy between design space exploration and simulation to enable the deployment configuration synthesis for automotive service-oriented architectures. Gerasimou et al. [39] presented a search-based approach named EvoChecker to synthesize the Pareto-optimal set of probabilistic models associated with the QoS requirements, which can support the selection of suitable system architecture and configuration. Although the above methods are promising in synthesizing different software artifacts, few of them considered the timing behaviors of systems.

As an effective means to accurately figure out the interaction between software artifacts and their surrounding physical environment, reactive synthesis enables the design of correct-by-construction reactive systems from their temporal logic specifications [4]. For example, based on fluent LTL, D'Ippolito et al. [40] presented a novel technique for synthesizing behavior models that works for an expressive subset of liveness properties and conforms to the foundational requirements engineering World/Machine model. Seshia and Subramanyan [41] presented a new system named UCLID5 for formal modeling, verification, and synthesis, which can model heterogeneous computational systems involving hardware, software, networking, and physical processes. Katis et al. [42] presented a novel Skolem extraction algorithm to enable the synthesis of witnesses with random behavior for various kinds of reactive systems (e.g., robot motion planning). Maoz and Ringert [43] presented a formal specification language named Spectra that is specifically tailored for use in the context of reactive synthesis. Meanwhile, in [44] they investigated 55 LTL specification patterns expressed in the GR(1) fragment of LTL (Linear Temporal Logic), and presented a translation of the LTL specification patterns to the GR(1) form for efficient reactive synthesis. In [45], Tadewos et al. developed a novel divide-and-conquer method for online Behavior Tree (BT) synthesis that can guide an autonomous agent to accomplish a series of missions expressed in Fragmented-Linear Temporal Logic (F-LTL). Although LTL-based reactive synthesis has been widely investigated, few of these works support timing behavior synthesis for UML-like requirements models. Moreover, due to high computational complexity, it is impractical to conduct reactive synthesis directly from LTL specifications.

Along with the increasing complexity of real-time embedded systems, CCSL becomes popular in the modeling [25], [46], analysis [47] and verification [21], [24], [48], [49] of timing behaviors of safety-critical software, e.g., automotive software [50], aerial control systems [7], and railway control systems [51]. However, most of the existing CCSL-based timing requirements analysis and verification methods assume CCSL specifications are complete and ready for processing, which is not always true in practice. To enable the automated generation of CCSL specifications, various CCSL synthesis methods have been investigated. For example, Hu et al. [27] tried to convert the CCSL synthesis process into a SKETCH problem, where the incomplete parts of CCSL specifications can be automatically figured out by underlying SAT/SMT solvers. Moreover, Hu et al. [26] treated the CCSL synthesis problem as a hole filling problem and proposed an efficient reinforcement learning-based heuristic, which can quickly form a complete CCSL specification under the help of various deductive techniques. Although these methods are promising in facilitating the CCSL specification generation, most of them are based on limited observations of system timing behaviors, while the safety properties of timing behaviors are neglected. Moreover, most of the existing CCSL synthesis methods focus on the generation of formal CCSL constraints rather than requirements models in semi-formal formats (e.g., MARTE, SysML). In other words, none of the existing CCSL synthesis methods can be directly used for the purpose of requirements model generation. To the best of our knowledge, our work is the first attempt that adopts CCSL as the IR to perform the synthesis of requirements models, whose timing behaviors satisfy the properties specified in safely-LTL formulas.

III. Preliminaries

In this section, we will detail the concepts, notations, and methods used in our safe timing behavior synthesis method.

A. Clock Constraint Specification Language

As a domain-specific specification language, CCSL adopts logical clocks to specify timing constraints of event orders.

Definition 3.1 (Logical Clocks): A logical clock c is a predicate over natural numbers. For any $i \in \mathbb{N}^+$, $c(i)$ indicates that if c is true at step i, c will tick at that step. Otherwise, c will idle at step i. ■

There are two types of logical clocks in CCSL, i.e., *atomic clocks* and *expression clocks*, where an atomic clock indicates a physical event in systems and an expression clock denotes an event involving existing clocks with certain relation. Table I presents all the operators defined in CCSL and their meanings. Generally, the operators can be classified into two categories based on relations between clocks. The operators in $O_b = \{=, \prec, \preceq, \subseteq, \#\}$ denote binary relations between two clocks, while the operators in $O_d = \{+, *, \wedge, \vee\}$ compose a new clock using existing clocks. Note that due to the space limitation, we do not give the formal definitions of such operators here. Please refer to [20] for more details.

CCSL specifications can be used to formalize the timing behaviors of various real-time embedded systems. Typically, a CCSL specification consists of a set of CCSL constraints, where each constraint is in one of the following forms:

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

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

$$c_e \triangleq c \ \$ \ d \ (on \ c_2), \text{ where } d \in \mathbb{N}^+ \tag{3}$$

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

TABLE I
CCSL Operators and Their Meanings

Operator	Syntax	Meanings
Causality	$c_1 \preccurlyeq c_2$	Clock c_1 always occurs greater than or equal to clock c_2.
Coincidence	$c_1 = c_2$	Clock c_1 always occurs equal to clock c_2.
Exclusion	$c_1 \# c_2$	Clock c_1 and clock c_2 will never occur simultaneously.
Precedence	$c_1 \prec c_2$	Clock c_1 always occurs strictly more than clock c_2.
Subclock	$c_1 \subseteq c_2$	Whenever c_1 occurs, it causes clock c_2 to occur.
Delay	$c_1 \triangleq c_2 \$ d$	Clock c_1 is delayed by clock c_2 with d time units.
DelayFor	$c_1 \triangleq c_2 \$ d \; on \; c_3$	Clock c_1 is delayed by clock c_2 with c_3 occurring d times.
Infimum	$c_1 \triangleq c_2 \wedge c_3$	Clock c_1 is the fastest clock but slower than clocks c_2, c_3.
Intersection	$c_1 \triangleq c_2 * c_3$	Whenever both clocks c_2 and c_3 occurs, clock c_1 occurs.
Periodicity	$c_1 \triangleq c_2 \propto p$	Clock c_1 periodically occurs with clock c_2 every p times.
Supremum	$c_1 \triangleq c_2 \vee c_3$	Clock c_1 is the slowest clock but faster than clocks c_2, c_3.
Union	$c_1 \triangleq c_2 + c_3$	Whenever either clock c_2 or c_3 occurs, clock c_1 occurs.

For example, the constraint $put \prec fetch$ specifies that the *fetch* operation on buffers must always follow the *put* operation, i.e., *put* must always occur strictly more than *fetch*. In CCSL, we define a schedule δ to describe the specific behavior of all the logical clocks.

Definition 3.2 (Schedule): Assuming that C is the set of logical clocks of a given CCSL specification, a schedule is a total function $\delta : \mathbb{N}^+ \rightarrow 2^C$ such that a clock $c \in C$ ticks at step $i \in \mathbb{N}^+$ if and only if $c \in \delta(i)$ and $\delta(i) \neq \emptyset$. ■

Note that a CCSL specification defines all the feasible schedules that conform to all its constraints. In CCSL, schedules are partially reflected by system traces, where a *trace* is a sequence of system events logged at their occurring time. We say that the CCSL specification admits a trace if the trace conforms to all its constraints.

B. CCSL Specification Synthesis

At the early stage of real-time embedded system design, due to various kinds of uncertain factors, requirements engineers often fail to write a full CCSL specification with all the timing information fixed. In this case, they usually leave such uncertainties as *holes* (indicated by the notation "??") in constraints, leading to incomplete specifications. Essentially, such holes are placeholders for either binary operators or clocks. An incomplete CCSL specification is the union of a set of regular CCSL constraints and a set of constraints with holes. For CCSL specification synthesis, it is required that an incomplete CCSL constraint contains only one hole for either a clock or an operator [26].

The *CCSL specification synthesis* is to automatically figure out contents for all the holes within incomplete CCSL constraints based on provided information (e.g., execution traces of systems). Given an incomplete CCSL specification S and a set of expected system traces T, the CCSL synthesis problem [27], [26] is to figure out a complete CCSL specification S' with all the holes filled, such that all the completed constraints in S' admit all the traces in T. Note that, so far, most CCSL specification synthesis methods are based on limited observations and expectations of system execution traces. Few of them support the synthesis under safety constraints.

C. Schedulability Analysis using MyCCSL

For a given set of complete CCSL constraints, designers have to tackle one fundamental issue known as *schedulability*, which decides whether there exists a schedule satisfying all the specified constraints. If the set of CCSL constraints is not schedulable, there should be some semantic conflicts among these constraints. To address such schedulability problem, MyCCSL [47], [52] is proposed as an efficient SMT-based schedulability analysis tool for CCSL. By taking a CCSL specification as the input, MyCCSL can check whether there exist schedules satisfying all its CCSL constraints within a limited bound (i.e., the number of steps). If the CCSL specification is schedulable, it means that there exists at least one feasible execution trace on which the occurrence of events strictly follows the constraints. Otherwise, it indicates that there are conflicts in the constraints, preventing all the events from occurring and resulting in some system deadlock. Meanwhile, MyCCSL is capable of verifying CCSL constraints against system properties in the form of LTL. By encoding both the CCSL constraints M and negated LTL-based properties P into SMT formulas, MyCCSL can check whether M satisfies P. If not, a counterexample will be generated, indicating an execution trace where the occurrence of events satisfies M but violates P.

D. LTL and Safely-LTL

As a formalism for specifying and verifying timing behaviors of reactive systems, LTL is defined as follows.

Definition 3.3 (LTL): Let $\mathcal{P}$ be a set of atomic propositions, and $a \in \mathcal{P}$ be an atomic proposition. Let X and U be two temporal operators, representing "Next" and "Until", respectively. The syntax of a linear temporal logic formula Φ is in the following format:

$$\Phi ::= \top \mid a \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid X\Phi \mid \Phi_1 U \Phi_2. \quad \blacksquare \tag{5}$$

Based on the semantics of logical and temporal operators in (5), we can derive the following operators: i) operator $\vee$ meaning "or", i.e., $\Phi_1 \vee \Phi_2 \equiv \neg(\neg\Phi_1 \wedge \neg\Phi_2)$; ii) operator $\rightarrow$ meaning "imply", i.e., $\Phi_1 \rightarrow \Phi_2 \equiv \neg\Phi_1 \vee \Phi_2$; iii) operator F meaning "Eventually", i.e., $F\Phi \equiv \top U\Phi$; and iv) operator G meaning "Always", i.e., $G\Phi \equiv \neg F\neg\Phi$. An LTL trace ρ in the form of "$\rho[1], \rho[2], \ldots$" is a sequence of propositional interpretations, where $\rho[n] \in 2^{\mathcal{P}} (n > 0)$. Note that, since we use LTL to specify the safety properties of timing behaviors of systems modeled by CCSL implicitly, in this paper each atomic proposition of LTL corresponds to one atomic clock in CCSL indicating a specific event. Given a trace ρ and an LTL formula Φ, we use $\rho, i \models \Phi$ to denote that Φ is *true* on ρ at step i. In this paper, we only use the safely-LTL for specifying the safety properties of timing behaviors of systems, which is a safety fragment of LTL defined as follows.

Definition 3.4 (Safely-LTL): An LTL formula Φ is a safely-LTL formula iff it does not contain an occurrence of F or U under an even number of the negation operator, i.e., "$\neg$". ■

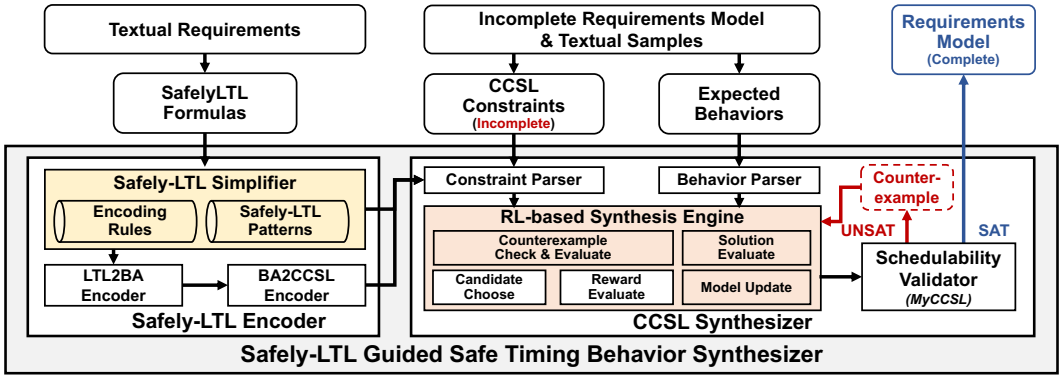


Fig. 1. Workflow of our synthesis approach

IV. Our Approach

To reduce the design complexity of timing behavior modeling of requirements models, we propose a novel synthesis method by resorting to CCSL, which enables the separation between the functional behavior modeling and timing behavior modeling of requirements models. Our goal is to help requirement engineers without the background of formal timing modeling to accurately and quickly construct requirements models satisfying specified safe timing requirements. Note that the concept of "incomplete" in our apporach means that some parts of the requirement models cannot be determined before synthesis. In other words, the requirement models have holes.

Figure 1 presents an overview of the workflow of our approach, which has three inputs: i) textual timing requirements specifying safety timing properties, ii) incomplete requirements models with unknown timing behaviors, and iii) textual samples specifying the expected timing behaviors of target systems. Since there exists a wide spectrum of methods and tools (e.g., *ARSENAL* [12]) that can extract LTL formulas from textual requirements, we do not detail how the safely-LTL formulas are derived here. Based on our proposed encoding rules and patterns, a complex safely-LTL formula can be decomposed into multiple simplified sub-formulas, where partial sub-formulas will be directly converted into CCSL constraints and the other sub-formulas will be transformed into Boolean Automata (BA) for CCSL constraint generation using *LTL2BA* [53] and *BA2CCSL* [46] encoders. Based on the existing tools (e.g., *RE4CPS* [54]), our approach can transform incomplete requirements models into their CCSL counterparts with holes.

Inspired by the state-of-the-art Reinforcement Learning (RL)-based CCSL synthesis engine named *CCSLRLSynthesizer* [55], we propose a novel RL-based synthesis approach. By combining the generated incomplete CCSL constraints from requirements models and the CCSL constraints from safely-LTL formulas, our approach can automatically figure out the unknown timing-related information of incomplete requirements models under the guidance of provided expected timing behaviors. Note that the CCSL constraints being completed by our RL-based CCSL synthesis engine may not be the desired ones to generate requirements models. This is mainly because the lengths of expected timing behaviors from textual requirements are limited, while the real timing behaviors of target systems are infinitely long. Therefore, our approach resorts to *MyCCSL* to validate the schedulability of the generated CCSL constraints. If the constraints cannot pass the validation, it means that the timing behaviors of synthesized CCSL constraints do not satisfy the specified safely-LTL formulas. In this case, our approach will use the synthesis result as a counterexample and feed it to the CCSL synthesizer for the next round of synthesis. The whole synthesis process will terminate if either the synthesis result succeeds in the validation or the synthesis reaches a timeout. For the former case, the complete version of the input incomplete requirements model will be automatically generated based on the synthesized CCSL constraints. Note that our approach uses CCSL as the IR for modeling and synthesis purposes, which is not touched by the requirements engineers. The following subsections will detail the key parts of our approach.

A. Safely-LTL-to-CCSL Encoding

As a safety fragment of LTL, safely-LTL [56] has been considered a promising way to model the safety properties of systems. Since safely-LTL can be automatically translated into CCSL constraints based on Boolean automata [46], our approach adopts it to specify the safe timing behaviors of expected requirements models. However, Boolean automata-based CCSL translation usually generates a large number of redundant clocks and constraints, since each automaton state needs to be modeled with one extra atomic clock while each automaton transition involves multiple CCSL constraints during the translation. As a result, the CCSL synthesis and validation time can be extremely long. To reduce such time, we present various safely-LTL-to-CCSL encoding rules and patterns as follows, which can significantly reduce the number of redundant clocks and CCSL constraints for a large set of safely-LTL formulas.

1) Encoding Rules: Assume that ϕ, ϕ_1, and ϕ_2 are all propositional logic expressions in the following formats:

$$p ::= a \mid \neg p \mid p \wedge p \mid p \vee p \mid p \rightarrow p \tag{6}$$

where a is an atomic proposition. During the safely-LTL-to-CCSL encoding, a propositional logic expression ϕ in some safely-LTL formula can be represented by a logical clock c in CCSL, indicating that the occurrences of timing behaviors satisfy ϕ in each step. If c ticks at step i, it means that the timing behavior satisfies ϕ at this step. To specify a safety property for system timing behaviors, one can use a safely-LTL formula in the form of $G(\neg\phi)$, which asserts that the behavior ϕ cannot happen in the target system. Definition 4.1 presents the satisfaction relations between the timing behaviors of target requirements models (represented by CCSL implicitly) and the specified safely-LTL-based timing requirements.

Definition 4.1 (Satisfaction of Safely-LTL Formulas): Let δ be a schedule of the CCSL specification S, and $G(\neg\phi)$ be a safely-LTL formula. The satisfaction $\delta \models G(\neg\phi)$ holds iff for any $i \in \mathbb{N}^+$ the propositional logic expression $\neg\phi$ is always true for the clock occurrences in $\delta(i)$. The satisfaction $S \models G(\neg\phi)$ holds iff $\delta \models G(\neg\phi)$ for any schedule δ of S. ■

To facilitate safely-LTL-to-CCSL encoding, we construct a global clock *glb*, which ticks if any clock in the system ticks. Assuming that C is the set of all system clocks, we define $glb \triangleq \sum_{clk \in C} clk$, where C is the set of all clocks in the target system and $\sum_{clk \in C} clk$ denotes the CCSL union of all the clocks in C. Note that based on *glb*, we can uniform the representations of CCSL schedules and safely-LTL traces for the purpose of timing behavior modeling. In other words, for any schedule δ of a CCSL specification S, we can have a counterpart safely-LTL trace for $G(\neg\phi)$ with the same form.

Rule 1: For the encoding of $G(\neg\phi)$, we use a logical clock c to denote the expression $\neg\phi$. By using only one CCSL constraint (i.e., $glb{=}c$), we can encode the propositional logic expression $\neg\phi$ in $G(\neg\phi)$ to its CCSL counterpart, i.e., c.

During the encoding, for each propositional logic expression ϕ, we propose a *recursive encoding function* $f(\phi)$ to convert ϕ into CCSL constraints using the following rules.

Rule 2: If ϕ is in the form of an atomic proposition $a \in \mathcal{P}$, we define $f(a) := c_a$[1], where the atomic clock c_a denotes whether system timing behaviors satisfy a in every step (i.e., $\forall i \in \mathbb{N}^+.\ \rho, i \models a$ iff $c_a \in \delta(i)$) assuming that ρ and δ are any safely-LTL trace and its corresponding CCSL schedule, respectively.

Rule 3: If the propositional logic expression is in the form of $\neg\phi$, we construct a new atomic clock $c_{\bar{\phi}}$ to denote whether timing behaviors satisfying $\neg\phi$ in every step, i.e., δ, where $\forall i \in \mathbb{N}^+.\ \rho, i \models \neg\phi$ iff $c_{\bar{\phi}} \in \delta(i)$. Besides the introduction to a new clock $c_{\bar{\phi}}$, The calculation of $f(\neg\phi)$ involves three new CCSL constraints and one more encoding function calculation for $f(\phi)$, i.e.,

$$c_\phi \ \#\ c_{\bar{\phi}},\ \ tmp \triangleq c_\phi + c_{\bar{\phi}},\ \ glb = tmp,\ \ f(\phi) := c_\phi,$$

where c_ϕ is a clock denoting the behavior of ϕ.

Since ϕ and $\neg\phi$ cannot be satisfied at the same step, c_ϕ and $c_{\bar{\phi}}$ cannot tick at the same step. Therefore, there is a CCSL exclusion relation between c_ϕ and $c_{\bar{\phi}}$, which means that in any step either c_ϕ or $c_{\bar{\phi}}$ must tick. The *tmp* is a union clock of c_ϕ and $c_{\bar{\phi}}$, which coincides with the global clock *glb*. Note that to figure out $f(\neg\phi)$, we need to calculate $f(\phi)$ in a recursive manner. The correctness of rule 3 can be proved as follows.

Proof According to the definition of c_ϕ in rule 2, we have

$$\forall i \in \mathbb{N}^+.\ \rho, i \models \phi \longleftrightarrow c_\phi \in \delta(i). \tag{7}$$

Based on the definition of $c_\phi \ \#\ c_{\bar{\phi}}$, we have

$$\forall i \in \mathbb{N}^+.\ c_\phi \notin \delta(i) \vee c_{\bar{\phi}} \notin \delta(i). \tag{8}$$

According to the definition of $tmp \triangleq c_\phi + c_{\bar{\phi}}$, we have

$$\forall i \in \mathbb{N}^+. tmp \in \delta(i) \longleftrightarrow\ \ c_\phi \in \delta(i) \vee c_{\bar{\phi}} \in \delta(i). \tag{9}$$

According to the definitions of *glb* and $glb = tmp$, we have

$$\begin{aligned} &\forall i \in \mathbb{N}^+.\ glb \in \delta(i), \\ &\forall i \in \mathbb{N}^+.\ tmp \in \delta(i) \longleftrightarrow\ \ glb \in \delta(i). \end{aligned} \tag{10}$$

[1]Here, the notation ":=" indicates that c_a is the clock generated for $f(a)$.

From Equations 9 and 10, we can get

$$\forall i \in \mathbb{N}^+.\ c_\phi \in \delta(i) \vee c_{\bar{\phi}} \in \delta(i), \tag{11}$$

and from Equations 8 and 11, we can get

$$\forall i \in \mathbb{N}^+.(c_{\bar{\phi}} \in \delta(i) \wedge\ \ c_\phi \notin \delta(i)) \vee (c_{\bar{\phi}} \notin \delta(i) \wedge\ \ c_\phi \in \delta(i)). \tag{12}$$

From Equations 7 and 12, we can conclude that

$$\forall i \in \mathbb{N}^+.\ \rho, i \models \neg\phi \longleftrightarrow c_{\bar{\phi}} \in \delta(i). \tag{13}$$

□

Rule 4: If the propositional logic expression ϕ is in the form of $\phi_1 \wedge \phi_2$, we need to construct a clock $c_\wedge$ to denote whether timing behaviors satisfy the formula $\phi_1 \wedge \phi_2$ in every step. The calculation of $f(\phi_1 \wedge \phi_2)$ involves one new CCSL constraint and two more calculations for both $f(\phi_1)$ and $f(\phi_2)$, respectively, i.e.,

$$c_\wedge \triangleq c_{\phi_1} * c_{\phi_2},\ \ f(\phi_1) := c_{\phi_1}, f(\phi_2) := c_{\phi_2},$$

where c_{ϕ_1} and c_{ϕ_2} are clocks to denote the behavior of formula ϕ_1 and ϕ_2, respectively.

Rule 5: If the propositional logic expression ϕ is in the form of $\phi_1 \vee \phi_2$, we need to construct a clock $c_\vee$ to denote whether timing behaviors satisfy the formula $\phi_1 \vee \phi_2$ in every step. Calculating $f(\phi_1 \vee \phi_2)$ introduces one new CCSL constraint and two calculations for $f(\phi_1)$ and $f(\phi_2)$, respectively, i.e.,

$$c_\vee \triangleq c_{\phi_1} + c_{\phi_2},\ \ f(\phi_1) := c_{\phi_1}, f(\phi_2) := c_{\phi_2},$$

Note that, since the proposition $\phi_1 \rightarrow \phi_2$ can be transformed into $\neg\phi_1 \vee \phi_2$, $f(\phi_1 \rightarrow \phi_2)$ can be calculated using rule 4.

Rule 6: For the *Next* operator X, to calculate $f(X\phi)$, we need to construct a clock $c_{X\phi}$ to denote that timing behaviors satisfy the formula $X\phi$ in every step. Calculating $f(X\phi)$ introduces 7 new CCSL constraints and one more calculation for $f(\phi)$, i.e.,

$$e_1 \triangleq c_{X\phi}\ \$\ 1\ on\ glb,\ e_1 \subseteq c_\phi,\ e' \triangleq c_{X\phi} + c_{\bar{X\phi}},\ e' = glb,$$
$$c_{X\phi}\ \#\ c_{\bar{X\phi}},\ e_2 \triangleq c_{\bar{X\phi}}\ \$\ 1\ on\ glb,\ e_2\ \#\ c_\phi,\ f(\phi) := c_\phi,$$

where the expression clock e_1 ticks one step after $c_{X\phi}$.

Since $X\phi$ means that ϕ is true in next step, when e_1 ticks, c_ϕ ticks in the same step. Note that, since c_ϕ may tick in the first step, the relation between e_1 and c_ϕ is *subclock* rather than *coincidence*. To ensure that $c_{X\phi}$ covers all the steps that satisfy $X\phi$, we define $c_{\bar{X\phi}}$ as the negated clock of $c_{X\phi}$ and make the next step of $c_{\bar{X\phi}}$ and c_ϕ exclusive. The correctness of rule 6 can be proved as follows.

Proof According to the definition of $e_1 \triangleq c_{X\phi}\ \$\ 1\ on\ glb$ and $e_1 \subseteq c_\phi$, we have

$$\forall i \in \mathbb{N}^+.\ e_1 \in \delta(i+1) \longleftrightarrow c_{X\phi} \in \delta(i), \tag{14}$$

$$\forall i \in \mathbb{N}^+.\ e_1 \in \delta(i) \longrightarrow c_\phi \in \delta(i). \tag{15}$$

Based on the definitions of both $e' \triangleq c_{X\phi} + c_{\bar{X\phi}}$ and $e' = glb$ as well as rule 2, we can get

$$\forall i \in \mathbb{N}^+.\ (c_{\bar{X\phi}} \in \delta(i) \wedge c_{X\phi} \notin \delta(i)) \vee (c_{\bar{X\phi}} \notin \delta(i) \wedge c_{X\phi} \in \delta(i)). \tag{16}$$

According to the definitions of $e_2 \triangleq c_{\bar{X\phi}}\ \$\ 1\ on\ glb$ and $e_2\ \#\ c_\phi$, we can get

$$\forall i \in \mathbb{N}^+.\ e_2 \in \delta(i+1) \longleftrightarrow c_{\bar{X\phi}} \in \delta(i), \tag{17}$$

$$\forall i \in \mathbb{N}^+.\ e_2 \notin \delta(i) \vee c_\phi \notin \delta(i). \tag{18}$$

From Equations 14 and 15, we can get

$$\forall i \in \mathbb{N}^+.\ c_{X\phi} \in \delta(i) \longrightarrow c_\phi \in \delta(i+1). \tag{19}$$

From Equations 16, 17, and 18, we have

$$\forall i \in \mathbb{N}^+.\ c_\phi \in \delta(i+1) \longrightarrow c_{X\phi} \in \delta(i). \tag{20}$$

From Equations 19 and 20, we can conclude

$$\forall i \in \mathbb{N}^+.\ \rho, i+1 \models \phi \longleftrightarrow \rho, i \models X\phi. \tag{21}$$

□

Note that our approach does not support the encoding function for safely-LTL formulas with operators F and U, which need to be translated to CCSL constraints based on Boolean automata.

Example: To illustrate how to encode a complex safely-LTL formula in the form of $G(a \rightarrow b)$ to its CCSL counterpart, Figure 2 show the major calculation steps for the encoding. By using our encoding rules, the safely-LTL generates six CCSL constraints, where *glb* denotes the union of all the clocks. Since this example has two clocks, i.e., c_a and c_b, we can get $glb \triangleq c_a + c_b$.

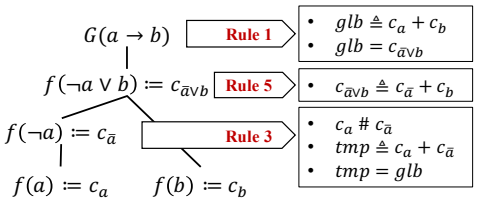


Fig. 2. An example of encoding $G(a \rightarrow b)$

2) Encoding Patterns: Compared with Boolean automata-based methods, the encoding rules proposed in Section IV-A1 can substantially reduce the number of generated CCSL constraints. However, these rules cannot be directly applied to safely-LTL formulas with operators F and U. Moreover, since the encoding rules are purely based on the syntax of safely-LTL formulas rather than their semantics, they still can lead to a large quantity of redundant CCSL constraints. To reduce the number of derived CCSL constraints as much as possible, we present four encoding patterns, which can be used to further optimize our rule-based encoding method.

Pattern 1: Let $G(\Phi_1 \wedge \Phi_2)$ be a safely-LTL formula involving two safely-LTL sub-formulas, i.e., Φ_1 and Φ_2. Based on the semantics of operator G, we can get $G(\Phi_1 \wedge \Phi_2) \equiv G(\Phi_1) \wedge G(\Phi_2)$ iff Φ_1 and Φ_2 are independent. It means that the encoding of $G(\Phi_1 \wedge \Phi_2)$ equals to the encoding of $G(\Phi_1)$ and $G(\Phi_2)$ in a separate manner. Let $Encoder_{ccsl}(\Phi)$ be a encoding function that can encode an arbitrary safely-LTL formula Φ to its CCSL counterpart. Based on the decomposition, the first encoding pattern is as follows:

$$\Phi_1 \wedge \Phi_2 \equiv Encoder_{ccsl}(\Phi_1) \cup Encoder_{ccsl}(\Phi_2)$$
$$G(\Phi_1 \wedge \Phi_2) \equiv Encoder_{ccsl}(G(\Phi_1)) \cup Encoder_{ccsl}(G(\Phi_2))$$

Let us take the encoding of $G(\Phi_1 \wedge \Phi_2)$ as an example. Assume that $G(\Phi_1)$ does not consists of operators F and U, while $G(\Phi_2)$ has. By adopting the decomposition in pattern 1, we can encode $G(\Phi_1)$ using our proposed rules and convert $G(\Phi_2)$ using Boolean automata-based methods [46]. In this way, compared with encoding based on Boolean automata only, the number of generated CCSL constraints using pattern 1 can be drastically reduced.

Note that even for the case where both $G(\Phi_1)$ and $G(\Phi_2)$ involve operators F or U, pattern 1 can also benefit the encoding to reduce the number of generated CCSL constraints by using the Boolean automata-based method. Figure 3 presents an example for the safely-LTL formula $G((a \rightarrow Gb) \wedge (c \rightarrow Gd))$, which can be decomposed into two sub-formulas, i.e., $G(a \rightarrow Gb)$ and $G(c \rightarrow Gd)$. Without using pattern 1, the Boolean automaton generated for $G((a \rightarrow Gb) \wedge (c \rightarrow Gd))$ has 4 states and 9 transitions, leading to 45 CCSL constraints. However, by using pattern 1, the encoding for the Boolean automata of $G(a \rightarrow Gb)$ and $G(c \rightarrow Gd)$ only generates 30 CCSL constraints.

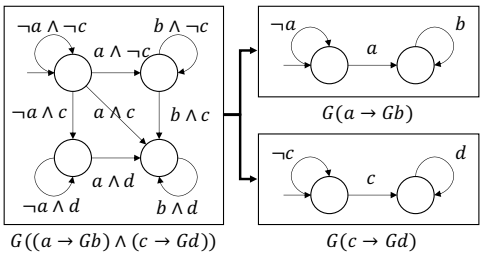


Fig. 3. An example of safely-LTL formula decomposition

When dealing with safely-LTL formulas in the form of $G(\phi)$, where ϕ is a propositional logic expression, we can use the following three encoding patterns to further reduce the number of generated CCSL constraints by resorting to the semantic equivalence relations between CCSL constraints. Note that the following patterns are not golden ones, where designers can fully exploit other semantic relations between CCSL notations to further reduce the number of generated CCSL constraints.

Pattern 2: If a safely-LTL formula is in the form of $G(\neg(\phi_1 \wedge \phi_2))$, where ϕ_1 and ϕ_2 are propositional logic expressions, we can encode the formula using the pattern:

$$c_{\phi_1} \# c_{\phi_2},\ f(\phi_1) \coloneqq c_{\phi_1},\ f(\phi_2) \coloneqq c_{\phi_2}.$$

As an example for the safely-LTL formula $G(\neg(a \vee b) \wedge \neg c)$, we can encode it into CCSL constraints "$e\#c,\ e = a + b$". By using this pattern, we do not need to construct the global clock or the negated clock for the CCSL constraint "$a + b$".

Pattern 3: If a formula is in the form of $G(\phi_1 \rightarrow \phi_2)$, where ϕ_1 and ϕ_2 are propositional logic expressions, we can encode the formula using the pattern:

$$c_{\phi_1} \subseteq c_{\phi_2},\ f(\phi_1) \coloneqq c_{\phi_1},\ f(\phi_2) \coloneqq c_{\phi_2}.$$

Since the safely-LTL formula $\phi_1 \rightarrow \phi_2$ equals to $\neg\phi_1 \vee \phi_2$, this pattern does not require to construct the global clock or the negated clock of c_{ϕ_1}.

Pattern 4: If a safely-LTL formula is in the form of $G((\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1))$, where ϕ_1 and ϕ_2 are propositional logic expressions, we can encode the formula using the pattern:

$$c_{\phi_1} = c_{\phi_2},\ f(\phi_1) := c_{\phi_1},\ f(\phi_2) := c_{\phi_2}.$$

Someone may find that the formula $G((\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1))$ can also be encoded based on patterns 1 and 3. Note that, by using pattern 4 the encoder only generates one *Coincidence* rule. However, by using the combination of patterns 1 and 3 the encoder will generate two *Subclock* rules. Since the validation cost of one *Coincidence* rule is typically less than that of the two *Subclock* rules, we suggest to use pattern 4 to encode the above safely-LTL formula in practice.

Example: For the safely-LTL formula $G((a \rightarrow b) \wedge ((\neg b \wedge \neg d) \vee \neg c))$, we can firstly use pattern 1 to decompose it to two sub-formulas, i.e., $G(a \rightarrow b)$ and $G((\neg b \wedge \neg d) \vee \neg c)$. For sub-formula $G(a \rightarrow b)$, by using pattern 3, we can obtain the CCSL constraint "$c_a \subseteq c_b$". For sub-formula $G((\neg b \wedge \neg d) \vee \neg c)$, by using pattern 2, we can get CCSL constraints "$e \triangleq c_b + c_d, e \,\#\, c_c$". Finally, by using our proposed encoding rules and patterns, we can encode the safely-LTL formula into three CCSL constraints. However, if none of the patterns are applied, we need to generate fourteen CCSL constraints base on our proposed encoding rules.

B. Safe Timing Behavior Synthesis

Inspired by the state-of-the-art CCSL synthesizer *CCSLRLSynthesizer* [26], [55], we propose a novel counterexample-guided CCSL synthesizer to generate requirements models with desired timing behaviors quickly. Our approach encodes the given incomplete CCSL specification as an RL model and uses expected timing traces with limited sizes to figure out an optimal solution. Since the synthesized results may not be the desired ones, our approach resorts to *MyCCSL* [52], which can validate the synthesis results by checking their schedulability under the given safely-LTL formulas. In an incremental way, if the validation fails, our approach will use the failed synthesis result as a counterexample and feed it into our synthesizer for a new round of synthesis. The synthesis process iterates until a complete and schedulable CCSL specification is found or the timeout triggers.

1) Counterexample Guided CCSL Synthesis: Reward plays an important role in determining the quality as well as the convergence speed of RL-based synthesis methods. Unlike *CCSLRLSynthesizer*, our RL-based CCSL synthesis adopts a novel reward evaluating mechanism by taking counterexamples into account, where a counterexample represents a set of unschedulable CCSL constraints. Note that the counterexamples here can come from different resources, e.g., the schedulability validator, textual requirements, and requirements engineers. At the end of each RL search in *CCSLRLSynthesizer*, the synthesis engine evaluates the rewards for each selected candidate, and fills all the incomplete constraints based on the selected candidates. Different from *CCSLRLSynthesizer*, our approach first checks whether the selected candidates match any recorded counterexamples. If yes, it will apply penalties on the candidates' rewards. Otherwise, our approach will conduct the same reward evaluation as that of *CCSLRLSynthesizer*.

Algorithm 1: Counterexample Guided Synthesis

Input: **i)** S, incomplete CCSL constraints; **ii)** T, expected timing traces; **iii)** S_c, counterexamples.
Output: A set of complete CCSL constraints.

```
1  SynthesisEngine(S,T,S_c) begin
2      π_θ → Init(S,T);
3      for i → 1 to k do
4          S' ← CandidateChoose(π_θ,T);
5          L_r → [];
6          if ConterexampleCheck(S',S_c) then
7              L_r → ConterexampleEvaluate(S',S_c);
8          else
9              L_r → RewardEvaluate(S',T,S_c);
10         end
11         π_θ → ModelUpdate(S',L_r,π_θ);
12     end
13     Spec ← SolutionEvaluator(S,π_θ);
14     return Spec;
15 end
```

Algorithm 1 details the implementation of our CCSL synthesis engine. Line 2 constructs an initial RL model based on the CCSL constraints S and expected traces T, where S is generated from the given incomplete requirements model and specified

safely-LTL formulas. Lines 3-12 conduct k rounds of RL training involving three major steps: i) searching for solutions admitting all the given expected traces; ii) calculating rewards for each solution; and iii) updating the RL model. In line 4, the function *CandidateChoose()* selects candidates to fill the holes in S such that the filled constraints can admit all the expected traces in T. The selection strategy is the same as [26]. In line 5, we use the list L_r to store the reward of each candidate. In line 6, the function *CounterexampleCheck()* checks whether the filled specification S' matches some counterexample in S_c. If yes, in line 7 the function *CounterexampleEvaluate()* will apply penalties (by default the value is -0.01) on the rewards of candidates in S' that match some counterexamples in S_c. Otherwise, in line 9 the function *RewardEvaluator()* will calculate the reward of each candidate based on the expected traces in T, which is same as *CCSLRLSynthesizer*. Therefore, the weights of candidates that lead to counterexamples in S_c can be reduced. In this way, our approach can effectively prevent the RL-based synthesis from selecting unschedulable candidates. Line 11 updates model π_θ based on the reward information in L_r. After k rounds of RL training, line 13 completes S using the trained RL model. Finally, line 14 returns the filled CCSL constraints.

2) CCSL Schedulability Validation: We adopt an SMT-based CCSL validator (i.e., *MyCCSL*) to check the schedulability of synthesized CCSL constraints. In our approach, a failed synthesized solution means that it cannot satisfy the given safety timing properties specified in safely-LTL formulas. This is because the RL-based synthesis process is guided by expected timing traces with limited sizes rather than unlimited ones. As shown in Section IV-B1, the synthesized solutions that failed in the validation can be fed back to the RL-based synthesis as counterexamples to prune the unfruitful search space. Note that, due to inherent conflicts between the given requirements models and specified safety timing requirements, the validation process may always fail before timeout. In this case, requirements engineers need to figure out the roots of such conflicts based on the accumulated counterexamples.

3) Our Safe Timing Behavior Synthesis Method: Algorithm 2 presents our safe timing synthesis algorithm in detail. In line 2, based on the rules and patterns proposed in Section IV-A, the function *SafelyLTLEncoder* encodes safely-LTL formulas in F_{ltl} into CCSL constraints. Line 3 combines S_{ltl} with the CCSL constraints derived from the incomplete requirements model RM. Line 4 uses the synthesis engine implemented in Algorithm 1 to complete S under the guidance of T. If such a complete version exists but does not pass the MyCCSL validation, it means that a counterexample is found. In this case, lines 6-7 will iteratively search for a feasible solution by using the synthesis method presented in Algorithm 1. If a solution is obtained, line 9 will return the elaborated requirements model based on the synthesis result.

Algorithm 2: Safe Timing Behavior Synthesis

Input: **i)** RM, an incomplete requirements model; **ii)** F_{ltl}, a set of safely-LTL formulas; **iii)** T, a set of expected timing traces; **iv)** S_c, a set of counterexamples.
Output: A synthesized requirements model.

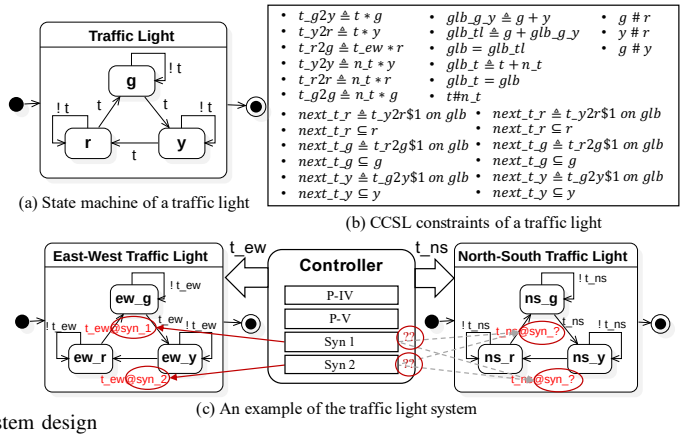
```
1  SafeLTLSynthesizer(RM,F_ltl,T,S_c) begin
2      S_ltl ← SafelyLTLEncoder(F_ltl);
3      S ← RMEncoder(RM) ∪ S_ltl;
4      Spec ← SynthesisEngine(S, T, S_c);
5      while Spec ≠ NULL ∧ MyCCSL(Spec) = UNSAT do
6          S_c ← S_c ∪ Spec;
7          Spec ← SynthesisEnginee(S, T, S_c);
8      end
9      return RMDecoder(Spec);
10 end
```

V. Experiments

To demonstrate the effectiveness of our CCSL synthesis method, we investigated two case studies, whose requirements models are built on top of two kinds of the most popular UML/MARTE models, i.e., statemachine diagrams and sequence diagrams. We developed the safely-LTL-to-CCSL encoder using the JAVA programming language, and implemented the counterexample-guided CCSL synthesis engine using the programming language Python. We adopted the open-source tool MyCCSL [52] to perform the schedulability validation of synthesized results. All the experiments were conducted on a laptop with Intel i9 2.4GHz CPU and 32GB memory.

A. Case I: Traffic Light Control System

The first case study is about a traffic light system to regulate traffic flow at a road intersection, which involves two traffic lights controlling the *North-South lanes* and *East-West lanes*, respectively. Here, each traffic light has three LED signals with green, yellow, and red colors, respectively. Note that at a time only one color is allowed to be shown by a traffic light, and the colors are shown following an order of green, yellow, and red in each iteration.



(a) State machine of a traffic light

(b) CCSL constraints of a traffic light

(c) An example of the traffic light system

Fig. 4. An example of traffic light system design

Figure 4(a) presents a UML statemachine diagram for the traffic light. It consists of three states, i.e., g, y, and r, indicating the states for green, yellow, and red colors, respectively, and t is a transfer signal to control transitions between states. Once the traffic light receives a transfer signal t, it will change its current state immediately. Based on the above description of the timing behaviors of traffic lights, we can derive the following properties:

P-I A traffic light shall never turn on two or more of its LEDs at the same time.

P-II A traffic light must change its current state immediately after receiving a transfer signal t.

P-III Without receiving any transfer signal t, a traffic light should not change its current state.

The three properties can be formalized using the following safely-LTL formulas:

$$G(\neg(g \wedge r) \wedge \neg(g \wedge y) \wedge \neg(r \wedge y)) \quad \textbf{(P-I)}$$

$$G(((g \wedge t) \rightarrow Xy) \wedge ((y \wedge t) \rightarrow Xr) \wedge ((r \wedge t) \rightarrow Xg)) \quad \textbf{(P-II)}$$

$$G(((g \wedge \neg t) \rightarrow Xg) \wedge ((y \wedge \neg t) \rightarrow Xy) \wedge ((r \wedge \neg t) \rightarrow Xr)) \quad \textbf{(P-III)}$$

Figure 4(b) presents the CCSL constraints for a traffic light, which are generated from both the statemachine diagram shown in Figure 4(a) and the specified properties I-III. These constraints can be used as the CCSL template of a traffic light to model its timing behaviors. Figure 4(c) shows the design of the traffic light system, which includes the E-W (East-West) traffic light, the N-S (North-South) traffic light, and a controller. Here, we use the controller to conduct the synchronization between the two traffic lights, where we use the clocks t_ew and t_ns to denote the transfer signals for the E-W traffic light and N-S traffic light, respectively.

Since traffic involves numerous vehicles, traffic signal control is a safety-critical system, which requires that vehicles in different directions cannot collide with each other. In other words, it is not allowed that E-W and N-S traffic can move simultaneously. To impose this constraint, we construct the following safety property:

P-IV: The N-S traffic light and the E-W traffic light cannot be in the state of green light or yellow light simultaneously. This property is formalized as the following safely-LTL formula:

$$G(\neg((g\_ns \vee y\_ns) \wedge (g\_ew \vee y\_ew))).$$

To regulate the intersection traffic in an efficient manner, a traffic light system needs to allow one traffic movement at any time. In other words, if one direction of the intersection is blocked, the other direction of the intersection should flow. To achieve this goal, we design the following property:

P-V: If the N-S traffic light is red, the E-W traffic light cannot turn red simultaneously, and vice versa. This property is formalized using the following safely-LTL formula:

$$G((r\_ns \rightarrow \neg r\_ew) \wedge (r\_ew \rightarrow \neg r\_ns)).$$

Based on the above safety properties, requirement engineers need to design a controller to regulate the intersection traffic via transfer signals. Since safely-LTL formulas of P-IV and P-V can be directly encoded into CCSL constraints, requirement engineers only need to figure out the synchronization implementation for t_ew and t_ns defined in statemachines. The annotation $@syn\_i$ in Figure 4(c) indicates that the associated transition in the current statemachine needs to be synchronized with a transition in the other state machine, where i is the synchronization index. The annotation $@syn\_?$ indicates that the counterpart transition for synchronization is unknown, where the synchronization index "?" needs to be figured out by the synthesizer. As shown in Figure 4(c), when a traffic light turns from red to green or from yellow to red, the traffic light of the other direction also needs to change accordingly at the same time. In Figure 4(c), the notations $@syn\_1$ and $@syn\_2$ in the left statemachine are used to enable the synchronization between two traffic lights. However, in the right statemachine such synchronization

information is unknown. As shown in Figure 4(c), the CCSL constraints of the controller are generated from two properties (i.e., P-IV and P-V) and two incomplete synchronizations (i.e., Syn_1 and Syn_2), where the incomplete synchronizations are encoded into CCSL constraints as follows:

$$e_{r2g} \triangleq t\_ew * r\_ew, e_{r2g} = \boxed{??},$$
$$e_{y2r} \triangleq t\_ew * y\_ew, e_{y2r} = \boxed{??}.$$

For both the incomplete requirements model shown in Figure 4(c) and specified safely-LTL properties, we generated 11 atomic clocks and 61 CCSL constraints (31 expression constraints and 30 relation constraints) by using our proposed encoding rules and patterns. We set the number of RL training rounds to 500 and set the maximum length of the traces that can be checked by the schedulability validator to 100. To guide the synthesis, we generated 5 timing traces with a length of 50 each, which describe the behaviors of different traffic scenarios. Figure 5 shows the synthesis result, where the filled parts are in red, denoting that the transition from *ns_r* to *ns_g* is involved in Syn_2 and the transition from *ns_y* to *ns_r* is involved in Syn_1. The whole synthesis process costs a total of 15.40s, including 2.79s for the synthesis and 12.61s for the schedulability validation.

- $t\_ew\_g2y \triangleq t\_ew * ew\_g$
- $t\_ew\_y2r \triangleq t\_ew * ew\_y$
- $t\_ew\_r2g \triangleq t\_ew * ew\_r$
- $t\_ew\_y2y \triangleq n\_t\_ew * ew\_y$
- $t\_ew\_r2r \triangleq n\_t\_ew * ew\_r$
- $t\_ew\_g2g \triangleq n\_t\_ew * ew\_g$
- $glb\_ew0 \triangleq ew\_g + ew\_y$
- $glb\_ew \triangleq ew\_g + glb\_ew0$
- $glb\_ew = glb$
- $glb\_t\_ew \triangleq t\_ew + n\_t\_ew$
- $glb\_t\_ew = glb$
- $t\_ew \# n\_t\_ew$
- $ew\_g \# ew\_r$
- $ew\_y \# ew\_r$
- $ew\_g \# ew\_y$
- $next\_t\_ew\_r \triangleq t\_ew\_y2r\$1\ on\ glb$
- $next\_t\_ew\_r \subseteq ew\_r$
- $next\_t\_ew\_g \triangleq t\_ew\_r2g\$1\ on\ glb$
- $next\_t\_ew\_g \subseteq ew\_g$
- $next\_t\_ew\_y \triangleq t\_ew\_g2y\$1\ on\ glb$
- $next\_t\_ew\_y \subseteq ew\_y$
- $next\_t\_ew\_r \triangleq t\_ew\_y2r\$1\ on\ glb$
- $next\_t\_ew\_r \subseteq ew\_r$
- $next\_t\_ew\_g \triangleq t\_ew\_r2g\$1\ on\ glb$
- $next\_t\_ew\_g \subseteq ew\_g$
- $next\_t\_ew\_y \triangleq t\_ew\_g2y\$1\ on\ glb$
- $next\_t\_ew\_y \subseteq ew\_y$

(a) CCSL constraints of E-W traffic light

- $t\_ns\_g2y \triangleq t\_ns * ns\_g$
- $t\_ns\_y2r \triangleq t\_ns * ns\_y$
- $t\_ns\_r2g \triangleq t\_ns * ns\_r$
- $t\_ns\_y2y \triangleq n\_t\_ns * ns\_y$
- $t\_ns\_r2r \triangleq n\_t\_ns * ns\_r$
- $t\_ns\_g2g \triangleq n\_t\_ns * ns\_g$
- $glb\_ns0 \triangleq ns\_g + ns\_y$
- $glb\_ns \triangleq ns\_g + glb\_ns0$
- $glb\_ns = glb$
- $glb\_t\_ns \triangleq t\_ns + n\_t\_ns$
- $glb\_t\_ns = glb$
- $t\_ns \# n\_t\_ns$
- $ns\_g \# ns\_r$
- $ns\_y \# ns\_r$
- $ns\_g \# ns\_y$
- $next\_ew\_r \triangleq t\_ns\_r2r\$1\ on\ glb$
- $next\_ew\_r \subseteq ns\_r$
- $next\_ew\_g \triangleq t\_ns\_g2g\$1\ on\ glb$
- $next\_ew\_g \subseteq ns\_g$
- $next\_ew\_y \triangleq t\_ns\_y2y\$1\ on\ glb$
- $next\_ew\_y \subseteq ns\_y$
- $next\_ns\_r \triangleq t\_ns\_r2r\$1\ on\ glb$
- $next\_ns\_r \subseteq ns\_r$
- $next\_ns\_g \triangleq t\_ns\_g2g\$1\ on\ glb$
- $next\_ns\_g \subseteq ns\_g$
- $next\_ns\_y \triangleq t\_ns\_y2y\$1\ on\ glb$
- $next\_ns\_y \subseteq ns\_y$

(b) CCSL constraints of N-S traffic light

- $ew\_g\_y \triangleq ew\_g + ew\_y$
- $ns\_g\_y \triangleq ns\_g + ns\_y$
- $ew\_g\_y \# ns\_g\_y$
- $ew\_r \# ns\_r$
- $t\_ew\_y2r = \mathbf{t\_ns\_r2g}$
- $t\_ew\_r2g = \mathbf{t\_ns\_y2r}$
- $glb \triangleq glb\_ew + glb\_ns$

(c) CCSL constraints of Controller

Fig. 5. Synthesized CCSL constraints for case 1

To validate the behaviors of synthesized CCSL constraints in Figure 5, we simulated the synthesized requirements model for ten times using the CCSL simulator *TimeSquare* [25], where Figure 6 presents the timing diagram of one simulation. Due to the limited space, this figure only presents the results within 48 ticks for the green lights, yellow lights, and red lights of both E-W traffic light and N-S traffic light. From this figure, we can observe that the synthesized requirements model satisfies both the given safety timing properties (i.e., **P-I-P-V**) and five expected timing behaviors (i.e., timing traces).

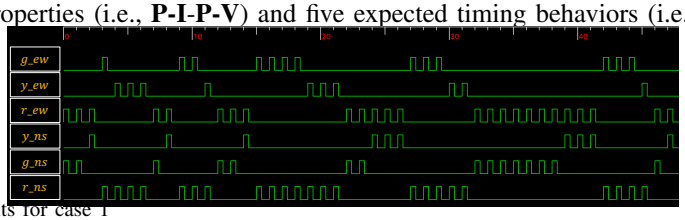


Fig. 6. Validation of synthesized results for case 1

B. Case II: WSP Mode Switching System

In the second case study, we investigated timing behavior synthesis for the requirements model of the Wayside Signal Protection (WSP) mode switching component within a train control system. Under WSP mode, the Vehicle On-Board Controller (VOBC) monitors both the speed and direction of a train in real-time, and detects the information on the wayside annunciator to prevent the train from speeding.

As shown in Figure 7, the case study is modeled in a UML sequence diagram by our industrial collaborator Casco Signal Ltd., showing the interactions between all the entities involved in WSP mode switching. In this requirements model, once a driver presses the WSP mode switching button, the VOBC will check whether it is in a healthy state (message 2). Then, the

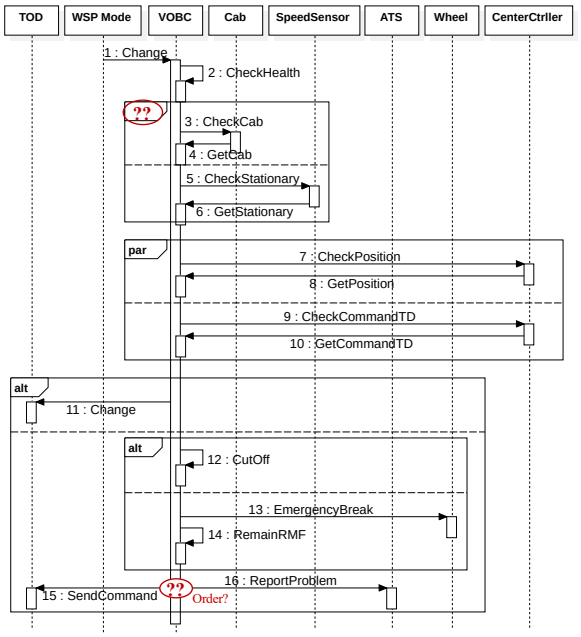


Fig. 7. Sequence diagram of WSP mode switching

VOBC will retrieve the status of the cab (messages 3 and 4) and the speed of the train (messages 5 and 6). Moreover, VOBC needs to figure out the position (messages 7 and 8) and command TD (messages 9 and 10) of the center controller. After that, if the train status is normal, VOBC will switch the mode and make a change on TOD (message 11). Otherwise, VOBC will activate an emergency brake (messages 12, 13, and 14) and report the problem to both ATS (message 16) and TOD (message 15).

By using the tools presented in [57], [54], the sequence diagram-based requirements model can be encoded to CCSL constraints automatically. Note that this sequence diagram is incomplete, where the type of the first fragment is unknown and the order of messages 15 and 16 is undetermined. The uncertain part of the first fragment can be encoded as incomplete CCSL constraints as follows:

$$FBegin \triangleq VOBC\_CheckCab\_snd \ \boxed{??} \ VOBC\_CheckStationary\_snd$$
$$FEnd \triangleq VOBC\_GetCab\_rev \ \boxed{??} \ VOBC\_GetStationary\_rev,$$

where the expression clock $FBegin$ and $FEnd$ denote the beginning and the end of a sequence diagram fragment, respectively. The type of this fragment depends on the expression operator type of $FBegin$ and $FEnd$. Generally, the undetermined order of two messages can be encoded using a CCSL constraint in the form of "$msg1\_snd??msg2\_snd$". Therefore, the undetermined order of messages 15 and 16 is encoded as follows:

$$VOBC\_ReportProblem\_snd \ \boxed{??} \ VOBC\_SendCommand\_snd.$$

TABLE II
COMPARISON BETWEEN SAFELY-LTL ENCODING METHODS

Safely-LTL	Bound	BA [46]		w/o Pattern		w/ Pattern	
		C. #	Time (s)	C. #	Time (s)	C. #	Time (s)
$G(a \rightarrow b)$	100	9	6.85	6	0.77	1	0.4
	500		232		52.94		6.24
$G(\neg a \vee \neg b)$	100	12	7.86	9	0.97	1	0.4
	500		373.9		67.1		1.95
$G((\neg a \wedge \neg b) \rightarrow (c \vee (\neg d \wedge \neg e)))$	100	18	19.29	15	10.62	4	0.87
	500		1359.4		352.7		93.87
$G((a \rightarrow b) \wedge (a \rightarrow c))$	100	10	9.15	7	2.92	2	0.69
	500		643.32		96.5		13.06
$G((a \rightarrow b) \wedge (b \rightarrow c))$	100	15	9.86	12	4.33	2	0.65
	500		663.14		199.6		8.62
$G((a \rightarrow (b \rightarrow (c \rightarrow d))))$	100	19	18.84	16	3.78	3	0.65
	500		1186.5		209		43.49
$G((a \rightarrow b) \wedge (a \vee c) \wedge (b \rightarrow (c \wedge d)))$	100	21	14.48	18	2.73	8	2.29
	500		1448.4		371.41		69.38

Note that the sequence diagram does not have selection conditions for *alt* fragments. According to the design manual of VOBC provided by Casco Signal Ltd., there are five preconditions for the WSP mode switching: i) the VOBC is healthy; ii) the train is stationary; iii) only one cab is active; iv) the position and orientation are established; and v) the command TD is determined. As shown in Figure 7, VOBC can send massage 11 if and only if these five preconditions are all satisfied. This requirement can be formalized as following properties:

P-I VOBC can switch to the WSP mode and send message *Change* to TOD only when the five preconditions holds, i.e., the VOBC is health ($VOBC.health = true$), the train is stationary ($stationary = true$), only one of the two cabs Cab_1 and Cab_2 is active ($(Cab_1 \vee Cab_2) \wedge \neg(Cab_1 \wedge Cab_2)$), the position and the orientation are established ($pos\_status = true \wedge ori\_status = true$), and the command TD is determined ($comm\_TD\_status = true$). This property can be formalized using the following safely-LTL formula:

$$G(VOBC\_Change\_snd \rightarrow (VOBC.health \wedge stationary \wedge((Cab_1 \vee Cab_2) \wedge \neg(Cab_1 \wedge Cab_2)) \wedge pos\_status \wedge ori\_status \wedge comm\_TD\_status)).$$

P-II If any precondition is not satisfied, VOBC will send a message to the other branch in the *alt* fragment. In this case, VOBC will send the message "*12:CutOff*" or "*13:EmergencyBreak*". This property can be formalized as the following safely-LTL formula:

$$G((VOBC\_CutOff\_snd \vee VOBC\_EmergencyBreak\_snd) \rightarrow \neg(VOBC.health \wedge stationary \wedge ((Cab_1 \vee Cab_2) \wedge \neg(Cab_1 \wedge Cab_2)) \wedge pos\_status \wedge ori\_status \wedge comm\_TD\_status))).$$

If the train status is abnormal, the mode switching will fail and VOBC needs to take emergency measures and report the problem. In this case, if the train is not stationary, VOBC must perform an emergent brake to stop the train. Otherwise, VOBC will immediately terminate the WSP switching. This requirement can be formalized using the properties as follows:

P-III If the train is stationary ($stationary = true$), VOBC will send the message "*12:CutOff*". This property can be formalized as the following safely-LTL formula:

$$G(VOBC\_CutOff\_snd \rightarrow stationary).$$

P-IV If the train is not stationary ($stationary = false$), VOBC will send the message "*13:EmergencyBreak*". This property can be formalized as the following safely-LTL formula:

$$G(VOBC\_EmergencyBreak\_snd \rightarrow \neg stationary).$$

In this case study, we use all the specified safely-LTL formulas (P-I to P-IV) to model the conditions of branch selection. From both the requirements model and the given safely-LTL formulas, this design generated 30 atomic clocks and 55 CCSL constraints (28 expression constraints and 27 relation constraints) in total by using our proposed encoding rules and patterns. For RL-based synthesis, we set the number of RL training rounds to 500, and set the maximum length of the traces that can be checked by the schedulability validator to 100. From both the design manual and engineers from Casco, we figured out 5 system timing traces with a length of 50 each to guide the timing behavior synthesis. To achieve a complete requirements model, our synthesis method costs 5372.78s, including 2.52s for CCSL synthesis and 5370.26s for the schedulability validation. The synthesis result is:

$$FBegin \triangleq VOBC\_CheckCab\_snd \vee VOBC\_CheckStationary\_snd$$
$$FEnd \triangleq VOBC\_GetCab\_rev \wedge VOBC\_GetStationary\_rev$$
$$VOBC\_ReportProblem\_snd \prec VOBC\_SendCommand\_snd.$$

From this result, we can fill the unknown fragment with *par*, and determine that message 16 is sent before message 15. We checked the completed requirements model with the engineers in Casco, and received the acknowledgment of synthesis correctness.

C. Comparison between Encoding Methods

Table II shows the comparison results of different safely-LTL encoding methods using seven widely used safely-LTL formulas. Note that in this experiment we focus on the encoding for safely-LTL formulas without taking functional behavior models into account. Since the schedulability validation time accounts for up to 99.9% (in case study II) of the overall requirements model synthesis time, here we only investigated the validation time for each safely-LTL formula with a specific bound by using MyCCSL [52]. Column 1 presents the seven safely-LTL formulas. Column 2 shows the maximum bound settings for MyCCSL. Column 3 has two subcolumns denoting the number of generated CCSL constraints and the validation time by using

the Boolean automata-based method, respectively. Here, we adopted the tool *LTL2BA* to convert the safely-LTL formulas into Boolean automata, and then used the method presented in [46] to encode the Boolean automata into CCSL constraints. Similarly, columns 4-5 present the results without and with using our proposed encoding patterns and encoding rules, respectively. From Table II, we can find that our pattern-based method generates the fewest number of CCSL constraints, which leads to the smallest validation time. As an example for the second safely-LTL formula, when the validation bound is 500, Boolean automata-based method generates 12 CCSL constraints costing 373.9 seconds, while our pattern-based encoding method only generates 1 CCSL constraint costing 1.95 seconds. Note that our proposed patterns can be combined to deal with the majority of commonly used safely-LTL formulas. Especially, by using pattern 1, we can decompose corresponding complex formulas in a divide-and-conquer manner. In the future, we plan to optimize our approach by investigating more effective and efficient patterns.

VI. Conclusion

Although CCSL has been acknowledged as an effective way to describe timing behaviors of safety-critical systems, it cannot be easily used by requirement engineers to form requirements models due to its complex syntax and semantics. To facilitate the requirements model generation without knowing preliminary CCSL knowledge, this paper presents a novel safe timing behavior synthesis method for requirement models. Based on our proposed safely-LTL-to-CCSL encoding rules and patterns, our approach can automatically transform both incomplete requirements models and corresponding safely-LTL-based timing properties into a set of CCSL constraints with holes. Guided by sampled timing behaviors, our approach can efficiently figure out the complete version of CCSL constraints, which can be used to construct the full requirements models satisfying both the given samples and safety timing properties. Comprehensive experimental results of two case studies show that our pattern-based safely-LTL-to-CCSL encoding method can derive fewer CCSL constraints compared with traditional automata-based methods, leading to significantly less CCSL synthesis and validation time. Moreover, the timing behaviors of generated requirements models by using our approach can indeed satisfy the specified safety properties.

Acknowledgments

This work was supported by the Natural Science Foundation of China (62272170), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, and "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software (22510750100). 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] D. Akdura, V. Garousib, and O. Demirörs, "A survey on modeling and model-driven engineering practices in the embedded software industry," *Journal of Systems Architecture*, vol. 91, pp. 62–82, 2018.
[2] M. Soeken, and R. Drechsler, "Formal Specification Level - Concepts, Methods, and Algorithms," *Springer*, 2015.
[3] Z. Wu, J. Liu, and X. Chen, "Better Development of Safety Critical Systems: Chinese High Speed Railway System Development Experience Report," *Proc. of Int. Conference on Automated Software Engineering (ASE)*, pp. 1216-1217, 2019.
[4] S. Maoz and J. Ringert, "On the Software Engineering Challenges of Applying Reactive Synthesis to Robotics," in *Proc. of International Workshop on Robotics Software Engineering (RoSE)*, pp. 17-22, 2018.
[5] M. Marques, E. Siegert, and L. Brisolara, "Integrating UML, Marte and SysML to improve requirements specification and traceability in the embedded domain," *International Conference on Industrial Informatics (INDIN)*, pp. 176-181, 2014.
[6] D. Amyot, A. A. Anda, M. Baslyman, L. Lessard, and J. Bruel, "Towards Improved Requirements Engineering with SysML and the User Requirements Notation," *International Requirements Engineering Conference (RE)*, pp. 329-334, 2016.
[7] 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 & Test in Europe Conference & Exhibition (DATE)*, pp. 376–381, 2020.
[8] A. Goknil, J. DeAntoni, M. Peraldi-Frati, and F. Mallet, "Analysis Support for TADL2 Timing Constraints on EAST-ADL Models," *Proc. of European Conference on Software Architecture (ECSA)*, pp. 89-105, 2013.
[9] A. M. Khan, F. Mallet, and M. Rashid, "Natural interpretation of UML/MARTE diagrams for system requirements specification," in *Proc. of IEEE Symposium on Industrial Embedded Systems (SIES)*, pp. 3193-198, 2016.
[10] Y. Wang, X. Lan, and Y. Wang, "Modeling Embedded Software Test Requirement Based on Marte," in *Proc. of International Conference on Software Security and Reliability (SERE) Companion*, pp. 109–115, 2013.
[11] E. Ebeid, F. Fummi, D. Quaglia, and F. Stefanni, "Refinement of UML/Marte models for the design of networked embedded systems," *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1072-1077, 2012.
[12] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner, "ARSENAL: Automatic Requirements Specification Extraction from Natural Language," in *Proc. of NASA Formal Methods: International Symposium)*, pp. 41–46, 2016.
[13] P. Nuzzo, M. Lora, Y. A. Feldman, and A. L. Sangiovanni-Vincentelli, "CHASE: Contract-based requirement engineering for cyber-physical system design," in *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 839–844, 2018.
[14] D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Rhein, J. Schumann, and N. Shi, "Formal requirements elicitation with FRET," in *Proc. of International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*, no. ARC-E-DAA-TN77785, 2020.
[15] N. G. Leveson, "Software safety in computer-controlled systems," *IEEE Computer*, vol. 17, no. 2, pp. 48-55, 1984.
[16] Object Management Group, "UML profile for MARTE: Modeling and analysis of real-time embedded systems," 2011.
[17] A. García-Domínguez, I. Medina-Bulo, and M. Marcos-Barcena, "Model-driven Design of Performance Requirements with UML and Marte," *Proc. of International Conference on Software and Data Technologies (ICSOFT)*, pp. 54-63, 2011.
[18] 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)*, pp. 167–176, 2009.

[19] 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)*, pp. 1–6, 2015.

[20] 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.

[21] 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)*, pp. 78–84, 2016.

[22] F. Mallet and R. De Simone, "Correctness issues on MARTE/CCSL constraints," *Science of Computer Programming*, vol. 106, pp. 78–92, 2015.

[23] 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)*, pp. 414–421, 2013.

[24] 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)*, pp. 61–70, 2017.

[25] 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*, pp. 34–41, 2012.

[26] M. Hu, J. Ding, M. Zhang, F. Mallet, and M. Chen, "Enumeration and Deduction Driven Co-Synthesis of CCSL Specifications Using Reinforcement Learning," *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, pp. 227-239, 2021.

[27] M. Hu, T. Wei, M. Zhang, F. Mallet, and M. Chen, "Sample-Guided Automated Synthesis for CCSL Specifications," *Proc. of Design Automation Conference (DAC)*, pp. 98:1-98:6, 2019.

[28] CIF, https://www.eclipse.org/escet/cif/.

[29] M. Goorden, L. Moormann, F. Reijnen, J. Verbakel, D. Beek, A. Hofkamp, J. Mortel-Fronczak, M. Reniers, W. Fokkink, J. Rooda, and L. Etman, "The Road Ahead for Supervisor Synthesis," in *Proc. of International Symposium on Dependable Software Engineering. Theories, Tools, and Applications (SETTA)*, pp. 1-16, 2020.

[30] O. Hussien and P. Tabuada, "Lazy Controller Synthesis using Three-valued Abstractions for Safety and Reachability Specifications," in *Proc. of IEEE Conference on Decision and Control (CDC)*, pp. 3567-3572, 2018.

[31] E. Letier and W. Heaven, "Requirements modelling by synthesis of deontic input-output automata," *Proc. of International Conference on Software Engineering (ICSE)*, pp. 592-601, 2013.

[32] S. Prabhu, G. Fedyukovich, K. Madhukar, and D. D'Souza, "Specification synthesis with constrained Horn clauses," in *Proc. of International Conference on Programming Language Design and Implementation (PLDI)*, pp. 1203-1217, 2021.

[33] Y. Li, Z. Xu, J. Cao, H. Chen, T. Ge, S. Cheung, and H. Zhao, "FlashRegex: Deducing Anti-ReDoS Regexes from Examples," *Proc. of International Conference on Automated Software Engineering (ASE)*, pp. 659-671, 2020.

[34] K. Wang, A. Sullivan, and S. Khurshid, "Automated Model Repair for Alloy," *Proc. of International Conference on Automated Software Engineering (ASE)*, pp. 1213-1217, 2018.

[35] M. Ceska, C. Hensel, S. Junges, and J. Katoen, "Counterexample-Driven Synthesis for Probabilistic Program Sketches," in *Proc. of International Symposium on Formal Methods (FM)*, pp. 101-120, 2019.

[36] B. Finkbeiner, H. Peter, and S. Schewe, "RESY: Requirement Synthesis for Compositional Model Checking," *Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 463-466, 2008.

[37] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. Varma, P. Kannan, A. Sivaraman, S. Narayana, and A. Gupta, "Switch Code Generation Using Program Synthesis," in *Proc. of Annual conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pp. 44-61, 2020.

[38] P. Obergfell, S. Kugele, and E. Sax, "Model-Based Resource Analysis and Synthesis of Service-Oriented Automotive Software Architectures," in *Proc. of International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 128-138, 2019.

[39] S. Gerasimou, R. Calinescu, and G. Tamburrelli, "Synthesis of probabilistic models for quality-of-service software engineering," *Automated Software Engineering*, vol. 25, pp. 785–831, 2018.

[40] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Synthesis of live behaviour models," in *Proc. of International Symposium on Foundations of Software Engineering (FSE)*, pp. 77-86, 2010.

[41] S. A. Seshia, and P. Subramanyan, "UCLID5: Integrating Modeling, Verification, Synthesis and Learning," *Proc. of International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pp. 1–10, 2018.

[42] A. Katis, G. Fedyukovich, J. Chen, D. Greve, S. Rayadurgam, and M. W. Whalen, "Synthesis of Infinite-State Systems with Random Behavior," *Proc. of International Conference on Automated Software Engineering (ASE)*, pp. 250-261, 2020.

[43] S. Maoz and J. Ringert, "Spectra: a specification language for reactive systems," *Software and Systems Modeling*, vol. 20, no. 5, pp. 1553-1586, 2021.

[44] S. Maoz and J. O. Ringert, "GR(1) synthesis for LTL specification patterns," in *Proc. of Joint Meeting on Foundations of Software Engineering (FSE)*, pp. 96-106, 2015.

[45] T. Tadewos, A. Newaz, and A. Karimoddini, "Specification-guided behavior tree synthesis and execution," *Expert Systems with Applications*, available online, 117022, 2022.

[46] R. Gascon, F. Mallet, and J. Deantoni, "Logical time and temporal logics: comparing UML Marte/CCSL and PSL," *Proc. of International Symposium on Temporal Representation and Reasoning (TIME)*, pp. 141-148, 2011.

[47] M. Zhang, F. Song, F. Mallet, and X. Chen, "SMT-based bounded schedulability analysis of the clock constraint specification language," in *Proc. of International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 61–78, 2019.

[48] 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 (ICFEM)*, pages 54–70, 2017.

[49] M. Larsen, "Verifying the Conformance of a Driver Implementation to the VirtIO Specification," in *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 719–720, 2021.

[50] E. Kang, D. Mu, and L. Huang, "Probabilistic verification of timing constraints in automotive systems using UPPAAL-SMC," in *Proc. of International Conference on Integrated Formal Methods (IFM)*, pp. 236–254, 2018.

[51] X. Chen, Z. Zhong, Z. Jin, M. Zhang, T. Li, X. Chen, and T. Zhou, "Automating consistency verification of safety requirements for railway interlocking systems," in *Proc. of Int. Requirements Engineering Conference (RE)*, pp. 308–318, 2019.

[52] MyCCSL, https://github.com/northcity0406/CCSLSMT.

[53] LTL2BA, http://www.lsv.fr/ gastin/ltl2ba/index.php.

[54] RE4CPS, *Requirements Engineering for Cyber-Physical Systems*, http://re4cps.org.

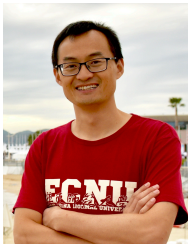
[55] CCSLRLSynthesizer, *Reinforcement Learning-Based CCSL Synthesizer*, https:// github.com/HMHelloWorld/CCSLRLSynthesizer.

[56] R. Lazic, "Safely Freezing LTL," in *Proc. of Int. Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pp. 381-392, 2006.

[57] 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*, vol. 216, pp. 102777, 2022.

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

Jun Xia (S'19) received the B.S. degree from the Department of Computer Science and Technology, Hainan University, Hainan, China, in 2016, and the M.E. degree from the Department of Computer Science and Technology, Jiangnan University, Wuxi, China, in 2019. He is currently working toward the Ph.D. degree with the Software Engineering Institute, East China Normal University, Shanghai, China. His research interests are in the areas of AIoT applications, trustworthy computing, and heterogeneous computing.



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.

Xiaohong Chen (M'16) received the Ph.D. degree from the Academy of Mathematics and Systems Science, Chinese Academy of Sciences, in 2010. She is currently an Associate Professor at East China Normal University. Her research interests include requirements engineering, model-driven development, and cyber-physical systems.

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 Aoste 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.

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 serves as the director of Engineering Research Center of Software/Hardware Co-design Technology and Application affiliated with the Ministry of Education, China, and the vice director of technical committee of embedded systems of China Computer Federation (CCF). 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.