Sample-Guided Automated Synthesis for CCSL Specifications

Ming Hu, Tongquan Wei, Min Zhang, Frédéric Mallet, Mingsong Chen

Sample-Guided Automated Synthesis for CCSL Specifications

Ming Hu[†], Tongquan Wei[†], Min Zhang[†], Frédéric Mallet[‡], Mingsong Chen[†]
[†]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai 200062, China
[‡] Université Cote d'Azur, CNRS, INRIA, I3S, France

ABSTRACT

The Clock Constraint Specification Language (CCSL) has been widely investigated in verifying causal and temporal timing behaviors of real-time embedded systems. However, due to limited expertise in formal modeling, it is difficult for requirement engineers to completely and accurately induce CCSL specifications from natural language-based design descriptions. To address this problem, we present a novel approach that facilitates automated synthesis of CCSL specifications under the guidance of sampled (expected) timing behaviors of target systems. By encoding sampled behaviors and incomplete CCSL constraints provided by requirement engineers using our proposed transformation templates, the CCSL specification synthesis problem can be naturally converted into a SKETCH synthesis problem, which enables the automated generation of CCSL specifications with high accuracy. Experiments on both well-known benchmarks and synthetic examples demonstrate the effectiveness and scalability of our approach.

1 INTRODUCTION

The mainstream design efforts for electronic design automation have moved from the *Register Transfer Level* (RTL) to the *Electronic System Level* (ESL) with high-level programming languages (e.g. SystemC) to deal with the complexity. ESL brings fast simulation, prototyping and design exploration techniques at early design stages. However, so far ESL itself does not significantly reduce the gap between high-level textual specifications and low-level synthesizable formalisms. The *Formal Specification Level* (FSL) [1] attempts to bridge this gap by introducing a formal, yet more abstract, layer for capturing unambiguously requirement specifications and guarantee the correct refinement down to lower levels. In this context, MARTE [2], a UML profile for *Modeling and Analysis of Real-Time and Embedded systems* attracts considerable attention, since it supports precise specification of both functional and non-functional system behaviors to be realized. In this paper, we focus on specification of timing behaviors of embedded systems.

The *Clock Constraint Specification Language* (CCSL) [3–5] is a companion language for MARTE. It gives a concrete syntax to handle logical time expressions that were popularized by Leslie Lamport. Logical clocks becomes first class citizens that model the repetitive events of a system. CCSL proposes a set of predefined clock constraints to express formal requirements using a library of predefined patterns that capture classical causal and temporal relationships among events. CCSL has effectively been used at FSL [6, 7] to generate SystemC code or monitor system executions. Various simulation or exhaustive exploration techniques have been proposed to explore the properties of a CCSL specification (by transformation to automata [8], transition systems [9], and SMT encoding [10]). Although all these CCSL property checking approaches can be fully automated, most existing methods assume that CCSL specifications are ready or can be easily obtained before the checking. This is not always true in reality.

Within the classical top-down design flow for embedded systems, CCSL specifications are manually derived from textual requirement specifications by requirement engineers. This will cause a serious problem, since majority of requirement engineers have limited knowledge in identifying formal definitions of time, clocks to access time, and relation between them. Things become even worse when the complexity of embedded systems is skyrocketing. It is becoming more and more difficult for requirement engineers to explore all the possible global timing behaviors from a complex design. In most cases, CCSL specification generation only relies on a limited number of system timing behaviors, resulting in inaccurate or incomplete view of the whole system. Consequently, the CCSL specification generation process is becoming time-consuming and particularly prone to errors. Although existing simulation and verification-based approaches can effectively detect CCSL specification errors, most of them focus on the ready-made CCSL specifications rather than the generation of CCSL specifications starting from scratch or semi-finished constraints. Clearly, the major bottleneck at this stage is a lack of methodologies and supporting tools that can facilitate the accurate CCSL specification generation for requirement engineers with limited expertise in formal timing modeling.

To enable automated generation of specifications, various specification synthesis approaches under the guidance of counterexamples or oracles were proposed [11, 12]. Aiming to bridge the gap between high-level specifications and low-level implementations, the synthesis methodology sketching [13] enables the automated generation of an implementation from the programmer provided sketch (i.e., partial program) by counterexample-guided inductive synthesis. Sketching has been successfully used in a variety of domains including scientific programs and concurrent data-structures. However, most of existing approaches focus on the synthesis of functional components, while few of them consider the synthesis of non-functional timing specifications.

Our contribution is inspired by the work introduced in [14] that employs sketching for JAVA GUI framework under the guidance of GUI event logs. In our approach, we assume that requirement engineers can partially figure out causal and temporal constraints (in terms of CCSL relations and expressions) among clocks, where the unknown parts are left as holes. Based on these incomplete CCSL constraints and the samples (expected timing behaviors), our approach can automatically synthesize CCSL specifications for target systems. This paper makes the **following three major contributions**: i) By encoding CCSL specification generation problems into sketching problems, we establish a SKETCH-based framework that enables requirement engineers to automatically synthesize formal timing behaviors. ii) We propose a set of effective priority policies for the synthesis of incomplete CCSL constraints, which can form CCSL specifications with high accuracy. iii) Based on our developed tool, we conduct various experiments on both CCSL benchmarks collected from the CCSL simulator TimeSquare [15] and complex synthetic examples to investigate the effectiveness of our approach in terms of synthesis accuracy and time. To the best of our knowledge, our approach is the first attempt that facilitates CCSL specification generation from scratch.

https://doi.org/10.1145/3316781.3317904

The rest of this paper is organized as follows. Section 2 introduces the notations used in CCSL synthesis. Section 3 proposes the details of our sample-guided CCSL synthesis approach. Section 4 presents the experimental results. Finally, Section 5 concludes this paper.

2 NOTATIONS FOR CCSL

CCSL relies on logical clocks to specify both partial event orders in distributed systems and synchronization conditions in synchronous languages. As defined in Definition 2.1, CCSL logical clocks are infinite sequences of ticks that do not have associated values.

DEFINITION 2.1. *A logical clock c is an infinite sequence of ticks.* ■

A CCSL schedule indicating a system timing behavior is an infinite sequence of steps, where each step defines which logical clocks can tick. A CCSL specification characterizes all the possible system behaviors.

DEFINITION 2.2. *Let C be a set of clocks. A schedule on C is a function* $\rho : \mathbb{N}^+ \rightarrow 2^C \backslash \{\emptyset\}$. ■

To model general clock constraints in CCSL, we adopt the notion of history that decides what may have at a given step. As defined in Definition 2.3, for a schedule ρ, $\chi_\rho(c,n)$ indicates the number of ticks of c before reaching step n. For the ease of expression, we use ρ^n and χ_c^n to denote $\rho(n)$ and $\chi_\rho(c,n)$ respectively when the context is clear. Assuming that the length of ρ is N, ρ can be represented as $\rho^1 \ldots \rho^N$.

DEFINITION 2.3. *The history of a schedule* ρ *over a clock set C is a function* $\chi_\rho : C \times \mathbb{N}^+ \rightarrow \mathbb{N}$, *such that for any* $c \in C$ *and* $n \in \mathbb{N}^+$:

$$\chi_\rho(c,n) = \begin{cases} 0 & n \leq 1 \\ \chi_\rho(c,n-1) & n > 1 \wedge c \notin \rho(n-1) \\ \chi_\rho(c,n-1)+1 & otherwise \end{cases} \quad ■ \qquad (1)$$

A CCSL specification consists of a set of constraints, where each constraint expresses the relations between the ticks of two clocks. Table 1 lists the eleven primitive CCSL operations that can be used to compose all the possible constraints [5], where the top five operators can model *relation constraints* and the remaining six operators can model *expression constraints*. It presents the definition of satisfiability of a schedule ρ against a constraint ϕ, denoted by $\rho \models \phi$.

Table 1: Semantics of CCSL Operators

Operators	ϕ	$\rho \models \phi$
Coincidence	$c_1 = c_2$	$\forall n \in \mathbb{N}^+ . \chi^n_{c_2} = \chi^n_{c_1}$
Precedence	$c_1 \prec c_2$	$\forall n \in \mathbb{N}^+ . \chi^n_{c_2} = \chi^n_{c_1} \Rightarrow c_2 \notin \rho^n$
Causality	$c_1 \preccurlyeq c_2$	$\forall n \in \mathbb{N}^+ . \chi^n_{c_1} \geq \chi^n_{c_2}$
Subclock	$c_1 \subseteq c_2$	$\forall n \in \mathbb{N}^+ . c_1 \in \rho^n \Rightarrow c_2 \in \rho^n$
Exclusion	$c_1 \# c_2$	$\forall n \in \mathbb{N}^+ . c_1 \notin \rho^n \vee c_2 \notin \rho^n$
Union	$c_1 \equiv c_2 + c_3$	$\forall n \in \mathbb{N}^+ . c_1 \in \rho^n \Leftrightarrow c_2 \in \rho^n \vee c_3 \in \rho^n$
Intersection	$c_1 \equiv c_2 * c_3$	$\forall n \in \mathbb{N}^+ . c_1 \in \rho^n \Leftrightarrow c_2 \in \rho(n) \wedge c_3 \in \rho^n$
Infimum	$c_1 \equiv c_2 \wedge c_3$	$\forall n \in \mathbb{N}^+ . \chi^n_{c_1} = max(\chi^n_{c_2}, \chi^n_{c_3})$
Supremum	$c_1 \equiv c_2 \vee c_3$	$\forall n \in \mathbb{N}^+ . \chi^n_{c_1} = min(\chi^n_{c_2}, \chi^n_{c_3})$
Delay	$c_1 \equiv c_2 \$ d$	$\forall n \in \mathbb{N}^+ . \chi^n_{c_1} = max(\chi^n_{c_2} - d, 0)$
Periodicity	$c_1 \equiv c_2 \propto p$	$\forall n \in \mathbb{N}^+ . c_1 \in \rho^n \Leftrightarrow c_2 \in \rho^n \wedge \exists m \in \mathbb{N}^+ . \chi^n_{c_2} = m \times p - 1$

In practice, it is impossible for requirement engineers to enumerate sample behaviors with inifinite lengths and random events. Therefore, our SKETCH-based approach only considers two kinds of schedules: i) the ones with finite lengths; and ii) the ones with infinite lengths but have periodic behaviors. Since a periodic schedule can be expressed in a regular-expression form, i.e., $\rho^1 \ldots \rho^i (\rho^{i+1} \ldots \rho^j)*$, in this case our approach uses the schedule with two repetitions (i.e., $\rho^1 \ldots \rho^j \rho^{i+1} \ldots \rho^j$) to guide the CCSL specification synthesis. In this way, both finite and infinite samples can be encoded and checked using a bounded schedule.

DEFINITION 2.4. *Let C and* Φ_C *be a clock set and a set of clock constraints on C, respectively. A feasible bounded schedule with a length of n satisfying* Φ_C *is a function* $\rho : \mathbb{N}^+_{\leq n} \rightarrow 2^C$ *such that* $\rho \models \phi$ *for every* $\phi \in \Phi_C$, *denoted by* $\rho \models \Phi_C$. ■

Due to limited samples provided by requirement engineers, we cannot always guarantee that a synthesized CCSL specification is a refinement of its golden reference. Therefore, Definition 2.5 presents a metric that can evaluate the accuracy of synthesized CCSL specifications based on bounded schedules.

DEFINITION 2.5. *A CCSL specification* Φ *is incomplete, if there exists a clock constraint* $\phi \in \Phi$ *that has holes indicating UNKNOWN clocks or CCSL opetators. Let* Φ^* *be the expected CCSL specification of a target system, and* Φ' *be a synthesized specification from* Φ *using sketching. Let* Σ_n *be the set of schedules* $\{\rho \mid length(\rho) = n \wedge \rho \models \Phi'\}$, *and* Σ'_n *be the set of schedules* $\{\rho \mid \rho \in \Sigma_n \wedge \rho \not\models \Phi^*\}$. *The synthesis accuracy with regard to schedules with a length of n is* $\frac{|\Sigma_n| - |\Sigma'_n|}{|\Sigma_n|}$. ■

To formalize priority policies using our approach, we define the *clock occurrence* and *clock slack* in Definition 2.6, and define the schedule refinement in Definition 2.7.

DEFINITION 2.6. *Let* ρ *be a finite schedule (with a length of n) of a CCSL specification on a clock set C. For any clock* $c \in C$, *the occurrence* $\Omega(\rho, c)$ *of c in* ρ *is a set of positive integers, i.e.,* $\{x \mid 0 < x \leq n \wedge c \in \rho(x)\}$. *The clock slack* $\delta(\rho, c)$ *indicates the number of steps that do not contain clock c in the tail of* ρ, *i.e.,* $\delta(\rho, c) = n - max(\Omega(\rho, c))$.

DEFINITION 2.7. *Let* ρ *be a schedule on clock set C . A relation operator X is a refinement of an operator Y, denoted by* $X \sqsupseteq Y$, *if for any* $c_1 \in C$ *and* $c_2 \in C$ *such that* $c_1 X c_2 \Rightarrow c_1 Y c_2$. ■

3 SKETCHING-BASED CCSL SYNTHESIS

Figure 1 presents the workflow of our sketching-based CCSL specification synthesis approach. The inputs are incomplete CCSL constraints and expected timing behaviors provided by requirement engineers. By using our CCSL parser, the relation and expression information collected from the incomplete CCSL constraints will be converted into a SKETCH implementation based on our proposed translation templates. By using our behavior parser, the expected timing behaviors will be translated into a SKETCH harness function. By combining both SKETCH implementations and harness functions, our approach encodes CCSL specification synthesis problems into sketching problems, which can be automatically solved by the synthesizer SKETCH [13]. Since SKETCH encoder is the kernel of our approach, the following subsections will introduce its major components (i.e., translation templates, relation encoder, expression encoder, harness encoder) in detail.

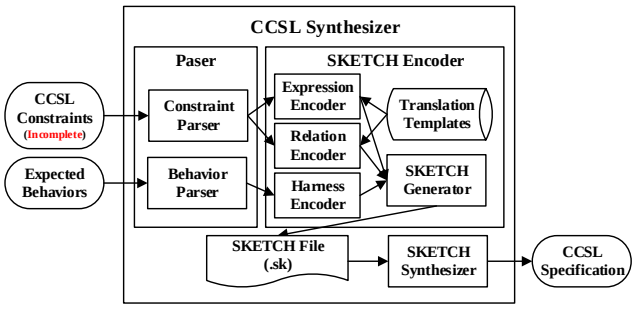


Figure 1: Workflow of our CCSL synthesis approach

3.1 Templates for CCSL Sketching

Typically, a SKETCH program consists of two parts, i.e., a SKETCH implementation and a harness. Our approach uses the SKETCH implementation to encode input incomplete timing constraints (i.e., incomplete

CCSL specification), and uses the harness to encode sampled (expected) timing behaviors for guiding the CCSL specification generation.

3.1.1 SKETCH Implementation Generation. Before introduction to the sketching of incomplete CCSL constraints, Figure 2 uses a sketching example for complete CCSL constraints to show how input CCSL constraints and the schedules extracted from sample behaviors are encoded as a sketching problem. Assume that the sampled timing behaviors have a length of 10. Based on Definition 2.4 and Definition 2.6, Figure 2(b) and Figure 2(c) show the schedule information parsed from sampled timing behaviors. In Figure 2(d), we classify the constraints into two categories based on the definitions in Table 1, i.e., expression constraints and relation constraints. As shown in Figure 2(a), the main body of our SKETCH implementation is a function that checks whether input schedules satisfy given CCSL constraints. In Figure 2(a), we use input parameters to encode the schedule ρ. For example, *c0_cnt* indicates the size of $\Omega(\rho, c_0)$, and c_0 saves the tick sequence of $\Omega(\rho, c_0)$. For each CCSL expression e_x ($x \in \{1,2\}$), we construct the data structures *ex_cnt* and *ex* in SKETCH to denote the length and occurrence of resultant intermediate clock for e_x. To achieve such information, our translation template module defines a comprehensive set of functions that cover all the operators based on the semantics presented in Table 1. As an example for expression e_0 in Figure 2(d), we use function *Intersection()* to calculate the intersection of the two input clock occurrences, and the size of the intersection is calculated by function *IntersectionCnt()*. For each relation constraint, we use one SKETCH function in the form of *checkX* to check whether the relation indicated by operator *X* is satisfied for the two associated logical clocks. For example, we use the function *checkPrecedence()* to check whether $c_0 \prec c_1$ is satisfied. If the input expected behavior violates any relation constraint, the *check()* function will return 0. Otherwise, *check()* will return 1 to indicate that the input timing behavior satisfies all the CCSL constraints.

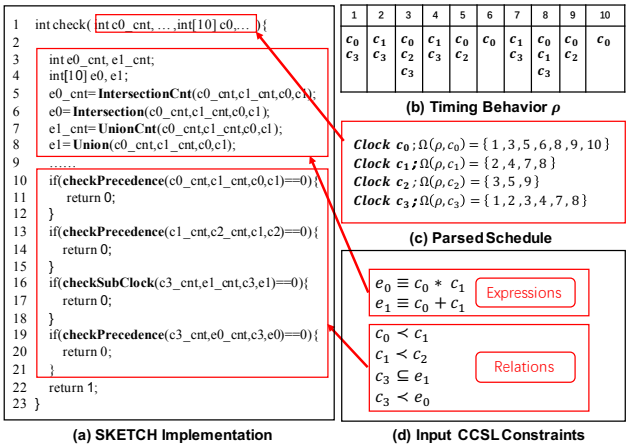


Figure 2: SKETCH generation for complete CCSL constraints

Although Figure 2 presents an overview of our SKETCH-based approach, the SKETCH implementations for complete constraints and incomplete constraints are quite different. Contrary to the example presented in Figure 2 that directly uses pre-defined SKETCH functions to check expression- or relation-based clock constraints, for different incomplete expressions or relations, our SKETCH encoder performs different encodings. As shown in Figure 3(b), the example has both incomplete expressions and relations, and the incomplete parts can be either clocks or operators. Similar to SKETCH, we use "??" to denote CCSL holes. For each incomplete relation, our relation encoder needs to rewrite its corresponding part shown in Figure 2. As an example for the relation "c_0??c_1" in Figure 3(b), we need to rewrite the part (lines 10-12) with the new encoding, since the relation operator cannot be figured out at this stage. The encoding for expressions is more complex than the one for relations. In our approach, since part of encoding for expression operator holes will be used by the encoding for expression clock holes, the encoding for incomplete expressions is performed as a whole (see Section 3.3). Note that the encoding for complete relations and expressions in Figure 3 is the same as the one shown in Figure 2.

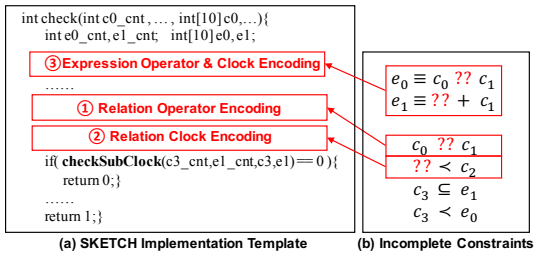


Figure 3: SKETCH generation for incomplete CCSL constraints

3.1.2 Harness Generation. The harness function is the main entrance of a C-like SKETCH program. Based on sample behaviors provided by requirement engineers, our CCSL synthesizer firstly parses the samples and translates them into corresponding schedules, and such schedules will be converted into a harness function to guide the CCSL specification generation. Listing 1 presents the template that we use for the generation of harness functions. In Listing 1, line 2 declares the data structures for saving sampled schedules. Note that our approach allows using multiple samples for guiding the synthesis, though this template only shows the use of two samples (indicated by lines 3-4 and lines 5-6, respectively). For each input sample, we adopt the SKETCH assertion to claim that it should satisfy the generated CCSL specification. In other words, the input samples are all feasible bounded schedules of our derived CCSL specification using sketching.

```
1  harness void main(){
2      int c0_cnt, ..., cn_cnt;  int[10] c0, ..., cn;
3      ......// initialize cx_cnt, cx (0<=x<=n) for the 1st sample
4      assert check(c0_cnt, ..., cn_cnt, c0, ..., cn) == 1;
5      ......// initialize cx_cnt, cx (0<=x<=n) for the 2nd sample
6      assert check(c0_cnt, ..., cn_cnt, c0, ..., cn) == 1;
7      ......
8  }
```

Listing 1: Template of generated harness function

3.2 Encoding for Incomplete Relations

3.2.1 Encoding for Relation Operators. When a hole represents a relation operator, our relation encoder will explore all the possible relation operators for it. To guarantee that the derived relation is accurate, we define the exploration priorities for relation operators as shown in Table 2 based on the refinement relations (see Definition 2.7) between relation operators. Due to the refinement sequence "=" ⊒ "⊆" ⊒ "≺" ⊒ "≼", we set coincidence relation with the highest priority and set the causality with the lowest priority. Since "#" has no refinement relation with other operators, we also set it with the highest priority.

Table 2: Priorities for Relation Operators

Operator	Coincidence =	Exclusion #	SubClock ⊆	Precedence ≺	Causality ≼
Priority	0	0	1	2	3

Listing 2 shows the SKETCH excerpt for constraint "c_0??c_1" defined in Figure 3(b). Due to the space limitation, we only present two of the five operator types here. For the operator hole, we use a variable *cond_0* to indicate the selected operator type for the hole. For example, if *cond_0* equals 0, the synthesized operator for the hole is "=". We set *cond_0* with the symbol "??" whose value will be figured out by SKETCH synthesizer. We use a variable *priority_0* together SKETCH built-in function *minimize()* to pose the operator selection priorities

during SKETCH-based synthesis. For example, if the check *checkCoincedence()* returns 1, the symbol "??" will be instantiated with "=" rather than other operators with lower priorities. Note that the current version of our SKETCH-based encoder implements all the nine kinds of operators that are supported by TimeSquare [1]. For example, "$\succcurlyeq$" is among the nine operators that can be constructed by the five primitive relation operators defined in Table 1.

```
1  int priority_0 = 0;
2  int cond_0 = ??;
3  assert cond_0>=0 && cond_0<9;
4  if(cond_0 == 0){
5     priority_0 = 0;
6     if(checkCoincedence(c0_cnt, c1_cnt, c0, c1)==0){return 0;}
7  }
8  else if(cond_0 == 1){
9     priority_0 = 2;
10    if(checkPrecedence(c0_cnt, c1_cnt, c0, c1)==0){return 0;}
11 } else ......
12 minimize(priority_0);
```

Listing 2: Encoding for the relation "$c_0??c_1$" in Figure 3(b)

3.2.2 Encoding for Relation Clocks. When synthesizing a relation clock marked with "??", we need to explore all the possible clocks to get a relation that leads to the highest synthesis accuracy. As an example shown in Figure 2(d), assuming that only the second relation is incomplete, i.e., "$?? \prec c_2$", we need to synthesize this constraint to approximate its original counterpart shown in Figure 2(d). Based on the parsed schedule presented in Figure 2(b), there are two choices, i.e., "$c_0 \prec c_2$" and "$c_1 \prec c_2$". Here, we prefer "$c_1 \prec c_2$", since it leads to a more accurate CCSL specification (with an accuracy of 1) than the one with "$c_0 \prec c_2$" (with an accuracy less than 1) . Based on our observation, when synthesizing the symbol "??" on the left-hand side of operator "$\prec$", the smaller occurrence cardinality a satisfying clock has, the higher accuracy the clock can achieve. Oppositely, when synthesizing the symbol "??" on the right-hand side of operator "$\prec$", a satisfying clock with the largest occurrence cardinality can achieve the highest accuracy. For example, if we want to synthesis the first relation in the form of '$c_0 \prec ??$" in Figure 2(d), c_1 is a better choice than c_2. Note that different relation operators have different preferences on clock occurrences for both sides.

Table 3: Priority for Relation Clock c_i

Operator	Coincid. =	Exclusion #	SubClock $\subseteq$	Precedence $\prec$	Causality $\preccurlyeq$
LPriority	0	|C| - rank(c_i)	|C| - rank(c_i)	rank(c_i)	rank(c_i)
RPriority	0	|C| - rank(c_i)	rank(c_i)	|C| - rank(c_i)	|C| - rank(c_i)

Assume that the input incomplete constraints contain m logical clocks and n expressions (i.e., temporary clocks). Let C denote the set of all these clocks. For a schedule ρ extracted from some expected behavior, we sort the $|C|$ clocks based on their occurrence cardinalities in an ascending order. Assuming that $c \in C$, we use *rank(c)* to denote the index of c in the sorted clock list, where *0≤rank(c)≤* |*C*|-1 and *rank(c)==0* indicates that c has the smallest occurrence cardinality. For the case of $\{c_i, c_j\} \subseteq C$, if $\Omega(\rho, c_i) == \Omega(\rho, c_j)$, we will sort them further based on their slack time. In this case, if $\delta(\rho, c_i) < \delta(\rho, c_j)$, we can get $rank(c_i) < rank(c_j)$. Based on the observation in last paragraph, Table 3 presents a priority table that can calculate the priority of $c_i \in C$ for the encoding of relation clocks. Note that all the relation operators have two operands. According to Table 3, the clock holes on different sides have different priorities. We use *LPriority* and *RPriority* to denote the priorities of clocks on the left-hand side and right-hand side, respectively.

Listing 3 shows an excerpt for encoding the incomplete relation "$?? \prec c_2$" in Figure 3(b). Unlike the encoding in Listing 2 that enumerates all possible operators, in this encoding we enumerate all the possible clocks for CCSL synthesizing, where the clock selection preference is determined by the priority as defined in Table 3. Since the operator in this case is $\prec$ and the unknown clock is on the left-hand side, we use pre-calculated *rank*[*i*] to indicate the *rank*(c_i) in Table 3.

```
1  ......//sort clocks based on occurrence and slack information
2  int priority_1 = 0;
3  int cond_1 = ??;
4  assert cond_1>=0 && cond_1<5;
5  if(cond_1 == 0){
6     priority_1 = rank[0];
7     if(checkPrecedence(c0_cnt, c2_cnt, c0, c2) ==0){return 0;}
8  }
9  else if(cond_1 == 1){
10    priority_1 = rank[1];
11    if(checkPrecedence(c1_cnt, c2_cnt, c1, c2) ==0){return 0;}
12 } else ......
13 minimize(priority_1);
```

Listing 3: Encoding for the relation "$?? \prec c_2$" in Figure 3(b)

3.2.3 Resolving Constraint Conflicts. When synthesizing multiple constraints, we may obtain the same relation constraints in the final specification, which are called *constraint conflicts*. For example, assuming that the incomplete constraint set contains "$c_0 \prec ??$" and "$?? \prec c_1$", we may get the result "$c_0 \prec c_1$" and "$c_0 \prec c_1$" for the two holes. To avoid this conflict, our relation encoder adopts a global conflict map based on hash map data structure that can check whether there exists a possible same relation constraint. Note that from Listing 2 and Listing 3 we can find that the value of variable *cond_i* indicates a special possible relation. During the SKETCH program generation, when our relation encoder enumerates a new possible constraint instance *const* indicated by "*cond_j = n*", it will query the conflict map. If in the conflict map there is no record with a key *const*, a new key-value pair <*const,{cond_j=n}*> will be added into the conflict map. Otherwise, if the constraint has been in the conflict map (e.g., indicated by <*const,{cond_i=m}*>), instead of enumerating the constraint using "*if(cond_j==n){...}*", we use the condition "*if(cond_j==n && cond_i!=m){...}*" in the SKETCH implementation to avoid duplicated constraints. Meanwhile, the item in <*const,{cond_i=m}*> conflict map will be replaced by <*const,{cond_i=m,cond_j=n}*>.

3.3 Encoding for Incomplete Expressions

3.3.1 Encoding for Expression Operators. Let C be a clock set and $\{c_1, c_2\} \subseteq C$. Let "$e = c_1??c_2$" be an incomplete expression constraint. Based on Table 1, we can find that e may have four possible forms, i.e., $c_1 + c_2$, $c_1 * c_2$, $c_1 \wedge c_2$, and $c_1 \vee c_2$, where $(c_0 + c_1) \preccurlyeq (c_0 \vee c_1) \preccurlyeq (c_0 \wedge c_1) \preccurlyeq (c_0 * c_1)$. Similar to the encoding of relation clocks, Table 4 sorts these four possible forms based on their occurrence cardinality information.

Table 4: Rank of Possible e with Different Expression Operators

Operator	Intersection $*$	Supremum $\wedge$	Infimum $\vee$	Union $+$
rank(e)	0	1	2	3

For input incomplete constraints, if there exists a relation that has an expression as its operand and the expression is incomplete in terms of expression operator, we need to adopt some priority policy to guide the expression operator synthesis. Note that expression itself is a temporary clock. Therefore, similar to the encoding in Section 3.2.2, Table 5 defines the priority of associated relation operators that affects the operator generation for possible e. For example, in Figure 3(b), the fourth relation has an operator "$\prec$". Since e_0 is on the right-hand side of "$\prec$", the priority for this case can be calculated using "$3 - rank(e)$". In other words, we prefer that $rank(e)$ equals 3. So the synthesizer will firstly try to synthesize e_0 with a union operator.

Table 5: Priority of Associated Relation Operators for Possible e

Operator	Coincid. =	Exclusion #	SubClock $\subseteq$	Precedence $\prec$	Causality $\preccurlyeq$
LPriority	0	3 - rank(e)	3 - rank(e)	rank(e)	rank(e)
RPriority	0	3 - rank(e)	rank(e)	3 - rank(e)	3 - rank(e)

Listing 4 presents the encoding for "$e_0 \equiv c_0??c_1$" in Figure 3(b). Since the relation "$c_3 \prec e_0$" affects the synthesis of symbol "??", we use the priority rule (see line 12 in Listing 4) defined in Listing 5 to generate the tightest specification.

```
1  int cond_e0 = ??;
2  assert cond_e0>=0 && cond_e0<4;
3  if(cond_e0 == 0){
4     e0_cnt = UnionCnt(c0_cnt, c1_cnt, c0, c1);
5     e0 = Union(c0_cnt, c1_cnt, c0, c1);
6     e0_op_priority = 3;
7  } else if(cond_e0 == 1){
8     e0_cnt = IntersectionCnt(c0_cnt, c1_cnt, c0, c1);
9     e0 = Intersection(c0_cnt, c1_cnt, c0, c1);
10    e0_op_priority = 0;
11 } else ......
12 int priority_3 = 3 - e0_op_priority;
13 if(checkPrecedence(c3_cnt, e0_cnt, c3, e0) ==0){return 0;}
14 minimize(priority_3);
```

Listing 4: Encoding for "$e_0 \equiv c_0??c_1$" in Figure 3(b)

3.3.2 Encoding for Expression Clocks. When synthesizing incomplete expressions, i.e., "$e_0 \equiv c_0+??$" and "$e_1 \equiv ??*c_2$", it is hard to decide which expression needs to be figured out first. If the hole in e_0 is filled with e_1, then e_0 depends on e_1. In other words, e_1 should be synthesized first. In this case, our expression encoder needs to figure out all the combinations of feasible expression dependencies before the SKETCH generation. In other words, our expression encoder needs to simultaneously encode all the expression constraints with unknown clocks together.

```
1  int exp_comb = ??;
2  assert exp_comb>=0 && exp_comb<3;
3  if(exp_comb == 0){
4     ......// insert lines 1-11 of Listing 4 here
5     e1_cnt = UnionCnt(c0_cnt, c1_cnt, c0, c1);
6     e1 = Union(c0_cnt, c1_cnt, c0, c1);
7  } else if(cond_e0 == 1){
8     ......// insert lines 1-11 of Listing 4 here
9     e1_cnt = UnionCnt(c2_cnt, c1_cnt, c2, c1);
10    e1 = Union(c2_cnt, c2_cnt, c2, c1);
11 } else if(cond_e0 == 2){
12    ......// insert lines 1-11 of Listing 4 here
13    e1_cnt = UnionCnt(e0_cnt, c1_cnt, e0, c1);
14    e1 = Union(e0_cnt, c1_cnt, e0, c1);
15 }
```

Listing 5: Encoding for "$e_1 \equiv ?? + c_1$" in Figure 3(b)

Listing 5 shows the encoding for the expression "$e_1 \equiv ?? + c_1$" in Figure 3(b) by our expression encoder. In this example, we consider all the possible cases for symbol "??", i.e., "$c_0 + c_1$", "$c_e + c_1$" and "$e_0 + c_1$". Note that "$c_3 + c_1$" is not considered by our expression encoder, since we have a relation "$c_3 \subseteq e_1$" such that "$c_3 \subseteq c_3 + c_1$" is trivial.

4 CASE STUDIES

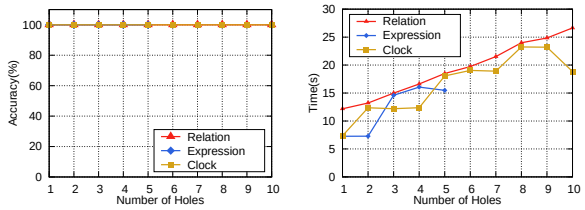
To evaluate the effectiveness of our approach, we conducted experiments on both CCSL specifications from the benchmark of CCSL simulation tool TimeSquare (version 1.0.0) [15] and complex synthetic specifications that are manually generated. We treated all these collected CCSL specifications without holes as the golden reference specifications for target systems. For each collected CCSL specification, we obtained its incomplete constraints by randomly removing clocks or operators from its relations and expressions. Note that for each relation or expression we only removed either one clock or one operator. Since for any given CCSL specification TimeSquare can generate random satisfying schedules with given lengths, we used it to produce the sampled timing behaviors of target systems. We developed our CCSL synthesizer (using JAVA programming language) which incorporates SKETCH (version 1.6.9) as our solving engine. All the experimental results were obtained on a Mac laptop with Intel i7 2.5GHz processors and 16GB RAM.

4.1 CCSL Synthesis Results

Table 6 presents the synthesis results for all the collected CCSL specifications. The first three columns show the sources of collection, specification names and CCSL operator types involved in specifications, respectively. Note that our experiments cover all the types of CCSL operators as shown in Table 1. For the case of Timesquare, each specification focuses on one specific operator type. For the case of synthetic examples, each specification consists of multiple relations and expressions with different operator types. The fourth column has three sub-columns showing the number of clocks, expressions and relations used in corresponding specifications, respectively. For each synthetic specification, we investigated two incomplete constraints. As shown in the fifth column, different sets of incomplete constraints have different holes on clocks, expression operators and relation operators. Note that for each specification, we adopted five sampled behaviors (each with a length of 50) to guide the synthesis. The sixth column denotes the overall CCSL specification synthesis time. To evaluate the accuracy of our approach, we randomly generate 200 schedules (each with a length of 200) from each newly generated CCSL specifications, and check them against the corresponding original specification. The seventh column shows the ratio of successfully passed schedules to the 200 schedules. The last column shows whether the generated CCSL specification is syntactically the same as its original counterpart. From this table, we can find that our approach can quickly and accurately figure out all the CCSL specifications. All the derived CCSL specifications are syntactically the same as their original counterparts except one (the second case of $Spec_3$). We checked the derived CCSL specification, and found that it is semantically equivalent to its original counterpart (i.e., $Spec_3$). Due to the limitation of space, we omit the proof here.

4.2 Impact Factors of Synthesis Accuracy and Time

This sub-section tries to figure out which factors strongly affect the accuracy and time of CCSL synthesis. Since holes are the major targets of CCSL synthesis, we firstly investigated the impacts of both the type and number of holes for incomplete constraints.



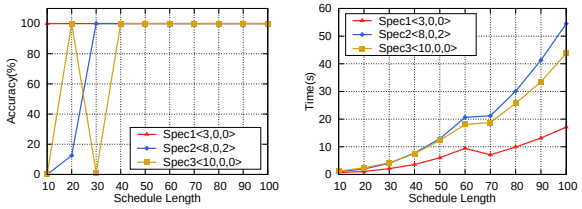
(a) Accuracy information (b) Time information

Figure 4: Impacts of the types and number of holes

Figure 4 shows the results of $Spec_3$ with different hole settings. In the sub-figures, different line colors indicate different types of holes that are uniquely applied on $Spec_3$. We use the red, blue and yellow colors to denote the incomplete constraint sets that only have relation operator-based, expression operator-based and clock-based holes, respectively. In this figure, the CCSL specification synthesis for each incomplete constraint set is guided by five random samples (each with a length of 50) generated by TimeSquare. From Figure 4(a), we can find that under the guidance of given samples our approach can derive CCSL specifications with 100% accuracy, even though every relation or every expression of $Spec_3$ has a hole. Since $Spec_3$ only has five expressions, we do not investigate the cases for $Spec_3$ with more than five expression operator-based holes. Figure 4(b) presents the synthesis time for incomplete constraint sets with different hole settings. We can find that generally more holes require more synthesis time. However, the synthesis time is not exponentially increased along with the increasing number of holes.

Table 6: Synthesis Results for TimeSquare Benchmarks and Synthetic CCL Specifications

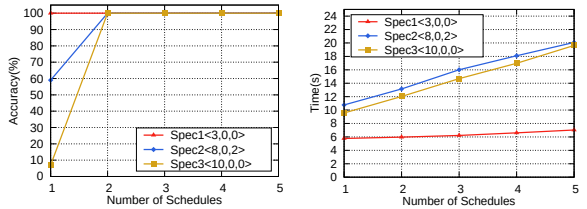
Source	Specification	Operator Type	Specification Statistics			Hole Settings			Time (ms)	Accuracy (%)	Same
			Clock	Expression	Relation	Clock	Expression Op.	Relation Op.			
TimeSquare	S	Coincidence	2	0	1	0	0	1	82	100	*yes*
	S_1	Precedence	2	0	1	0	0	1	510	100	*yes*
	S_2	Causality	2	0	1	0	0	1	512	100	*yes*
	S_3	Subclock	2	0	1	0	0	1	5019	100	*yes*
	S_4	Exclusion	2	0	1	0	0	1	4420	100	*yes*
	S_5	Union	3	1	1	0	1	0	1554	100	*yes*
	S_6	Intersection	3	1	1	0	1	0	2515	100	*yes*
	S_7	Infimum	3	1	1	0	1	0	2672	100	*yes*
	S_8	Supremum	3	1	1	0	1	0	2584	100	*yes*
	S_9	Delay	2	2	1	0	1	0	707	100	*yes*
	S_{10}	Periodicity	2	1	1	0	1	0	713	100	*yes*
Synthetic	$Spec_1$	Composite	4	1	3	3	0	0	6907	100	*yes*
						0	1	2	30961	100	*yes*
	$Spec_2$	Composite	10	0	10	0	0	10	24608	100	*yes*
						8	0	2	21468	100	*yes*
	$Spec_3$	Composite	10	5	10	10	0	0	20038	100	*no*
						4	2	4	32436	100	*yes*
	$Spec_4$	Composite	20	6	16	9	1	7	300198	100	*yes*
						5	1	10	32275	100	*yes*



(a) Accuracy information (b) Time information

Figure 5: Impacts of the lengths of schedules

Figure 5 investigates the impacts of trace lengths on synthesis accuracy and time. In the figure, we only show the results for three incomplete constraint sets generated from synthetic examples $Spec_1$, $Spec_2$ and $Spec_3$. In the figure, we use notation *Spec*<*x,y,z*> to denote different incomplete constraint sets. We use *Spec* to indicate the specification name presented in the second column of Table 6, and use <*x,y,z*> to indicate the hole settings for clocks, expression operators and relation operators. For example, $Spec_1 < 3,0,0 >$ denotes the first case of synthetic CCSL specification $Spec_1$ in Table 6. Note that, as shown in Table 6, our approach can achieve the highest synthesis accuracy using five random samples. To show the effects of samples with different lengths, the experiment for Figure 5 only use two traces (with different lengths) for specification synthesis. From Figure 5(a), we can find that the longer samples (i.e., schedules) requirement engineers can provide, the better synthesis accuracy we can achieve. Note that since all the samples with different lengths are randomly generated by Timesquare, longer samples may lead to worse synthesis accuracy. For example, when using two samples with a length of 30, the accuracy of the generated CCSL specification for $Spec_3 < 10,0,0 >$ is only 1%. However, when the length of the two samples is 20, the generated CCSL specification can achieve an accuracy of 100%. Figure 5(b) shows the synthesis time with difference sample lengths. We can observe an exponential trend of synthesis time when schedule lengths increase.



(a) Accuracy information (b) Time information

Figure 6: Impacts of the number of samples

Figure 6 shows the impacts of the number of schedules. In this experiment, we fix the sample length to 50. We can find that when more samples are involved in synthesis, the better CCSL specifications with higher accuracy can be achieved. From Figure 6(a), we can observe that by only two samples all the derived CCSL specifications can achieve the highest accuracy. Note that as shown in Figure 6(b) when more schedules are used to guide CCSL specification synthesis, the trend of synthesis time growth is more moderate than the one shown in Figure 5(b).

5 CONCLUSION

Although CCSL allows precise specification of timing behaviors, it is not widely accepted by requirement engineers in practical embedded system design. This is mainly because: i) most engineers lack expertise in formal modeling; and ii) no existing tools can automatically generate CCSL specifications for embedded systems with skyrocketing complexity. To alleviate this situation, this paper proposes a SKETCH-based CCSL specification synthesis approach, which facilitates requirement engineers to accurately and automatically build their timing designs. Experimental results on well-known benchmarks and synthetic examples show that our approach can effectively generate the tightest timing constraints under the guidance of expected behaviors of target systems.

ACKNOWLEDGMENTS

This research was supported by National Science Foundation of China (Grant No. 61872147). Mingsong Chen is the corresponding author.

REFERENCES

[1] R. Drechsler, M. Soeken and R. Wille. Formal Specification Level: Towards verification-driven design based on natural language processing. In *Proc. of FDL*, 53–58, 2012.
[2] Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, 2011.
[3] C. André and F. Mallet. Specification and verification of time requirements with CCSL and Esterel. In *Proc. of LCTES*, 167–176, 2009.
[4] 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 DAC*, 122:1–122:6, 2015.
[5] 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.
[6] E. Kang and P. Schobbens. Schedulability analysis support for automotive systems: from requirement to implementation. In *Proc. of SAC*, 1080–1085, 2014.
[7] J. Peters, R. Wille and R. Drechsler. Generating SystemC Implementations for Clock Constraints Specified in UML/MARTE CCSL. In *Proc. of ICECCS*, 116–125, 2014.
[8] J. Peters, N. Przigoda, R. Wille and R. Drechsler. Clocks vs. instants relations: Verifying CCSL time constraints in UML/MARTE models. In *Proc. of MEMOCODE*, 78–84, 2016.
[9] F. Mallet and R. Simone. Correctness issues on MARTE/CCSL constraints. *Science of Computer Programming*, vol. 106, pp. 78–92, 2015.
[10] 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 LCTES*, 61–70, 2017.
[11] A. Albarghouthi, I. Dillig and A. Gurfinkel. Maximal specification synthesis. In *Proc. of POPL*, 789–801, 2016.
[12] S. Jha, S. Gulwani, S. A. Seshia and A. Tiwari. Oracle-guided component-based program synthesis. In *Proc. of ICSE*, 215–224, 2010.
[13] A. Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 15(5-6), pp. 475–495, 2013.
[14] J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster and A. Solar-Lezama. Synthesizing framework models for symbolic execution. In *Proc. of ICSE*, 156–167, 2016.
[15] J. DeAntoni and F. Mallet. TimeSquare: Treat Your Models with Logical Time. In *Proc. of TOOLS*, 34–41, 2012.