A Generic Representation of CCSL Time Constraints for UML/MARTE Models

Judith Peters[1] Robert Wille[1,2] Nils Przigoda[1] Ulrich Kühne[1] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{jpeters,rwille,przigoda,ulrichk,drechsler}@informatik.uni-bremen.de

***Abstract*—The complexity of today's embedded and cyber-physical systems is rapidly increasing and makes the consideration of higher levels of abstraction during the design process inevitable. In this context, the impact of modeling languages such as UML and its profiles such as MARTE is growing. Here, CCSL provides a formal description of timing constraints which have to be enforced on the considered system. This builds the basis for many further design steps and can be used e. g. for checking the consistency of the specification, for code generation, or for proving whether the time constraints have correctly been implemented at lower abstraction levels. However, most of the approaches available thus far usually focus on sole design tasks only – often even without an explicit consideration of the system's functional behavior. In this work, we are aiming for overcoming this drawback by providing a method to automatically generate a generic representation of a set of clock constraints in terms of a transition relation. Afterwards, the resulting transition relation can easily be utilized for the above mentioned design tasks. A discussion on the applicability of the generic description as well as an exemplary evaluation shows the promise of the proposed generic representation.**

I. Introduction

Today's embedded systems are formed of up to millions of components such as gates, signals, and lines enriched by additional sensors and actors eventually forming cyber-physical systems. The increasing complexity of these systems makes the consideration of higher levels of abstractions in their design inevitable. Modeling languages such as the *Systems Modeling Language* (SysML, [1]) as well as the *Modeling and Analysis of Real-time and Embedded systems* profile (MARTE, [2]) find considerable attention in this regard. They allow for a very precise specification of the functional as well as non-functional behavior for a system to be realized.

In this contribution, we consider the specification of timing behavior. Timing is an essential part of a specification and particularly of interest in all cases where the question "when?" is asked. Specifying timing behavior is complex, because it needs to provide a clear definition of time, clocks to access time, and relations between all of them. For this purpose, MARTE provides the *Clock Constraint Specification Language* (CCSL, [2]) which allows for a precise specification of timing behavior in complex systems. This builds the basis for the next designs steps and can be used e. g. to (1) check whether the timing constraints are consistent, plausible, and indeed have been specified as intended, (2) generate code which actually realizes the desired timing behavior, as well as (3) proving whether the time constraints have correctly been implemented at lower abstraction levels. In the recent past, very powerful and complementary methods addressing these tasks have been proposed e. g. in [3], [4], [5], [6], and [7].

These methods, however, usually rely on a translation and interpretation of the CCSL descriptions in some different computational model or language. Generating this is a laborious and, at the same time, highly critical task in which any error will spoil the validity of the respective results. Thus far, researchers approached this challenge by providing translation schemes for single application scenarios only. For example, checking CCSL as introduced in [3], [4], [5], [6] relies on a direct encoding of the constraints e. g. into Promela- or UPPAAL-syntax. Code generation as proposed in [7] directly creates SystemC code. That is, for each design task usually a different translation and interpretation of CCSL constraints is applied – although all of them eventually shall rely on the same semantics.

Moreover, in most of the existing approaches, the non-functional behavior of the system is not explicitly considered. This constitutes a significant drawback since e. g. methods such as [8], [9], [10], [11], [12], [13] which are aiming for proving the correctness of corresponding models do not support CCSL constraints. Vice versa, CCSL checkers as introduced in [3], [4], [5], [6] do not consider the functional behavior. That is, functional and timing constraints are usually checked independently of each other. Consequently, design flaws caused by the combination of *both* constraints are often not considered before an initial implementation is available. Other design tasks such as code generation similarly suffer from the current state-of-the-art: For example, the realization of CCSL timing constraints in SystemC as described in [7] relies on the fact that the entire functional description of the system is already implemented.

Overall, the "inflation" of solutions for rather specific design tasks and, at the same time, their inability to combine them with tools for functional code generation and verification significantly limits the application of CCSL timing constraints in practice.

In this work, we aim for a generic representation of the CCSL constraints which (1) can be used for various purposes such as verification, code generation, and more as well as (2) can easily be integrated and merged with corresponding representations of the functional behavior of the system un-

DAC '15, June 07 - 11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3520-1/15/06…$15.00
http://dx.doi.org/10.1145/2744769.2744775

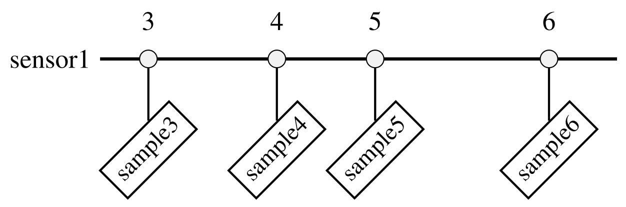


Fig. 1. An exemplary time line of a clock `sensor1`

der consideration. To this end, we propose a transformation scheme which takes CCSL constraints and maps those into an equivalent representation in terms of a transition relation. Afterwards, the resulting transition relation can easily be integrated into existing solutions for design tasks mentioned above.

The contribution is described in the remainder of this work as follows: Section II briefly reviews CCSL. Afterwards, the general idea of the generic representation (including a formal definition) as well as a methodology for its automatic generation are introduced in Section III and Section IV, respectively. Finally, Section V discusses the applicability of the generic description and provides an exemplary evaluation before the paper is concluded in Section VI.

II. Preliminaries

The *Modeling and Analysis of Real-time and Embedded systems* profile (MARTE) for UML provides a language for describing timing constraints: the *Clock Constraint Specification Language* (CCSL) [2]. A central part of the underlying time definition are *instants*, i. e. moments in the raw, unordered time, defined by clock ticks. The *clock* is an instrument to access a set of instants [14]:

Definition 1. *A* clock $c = \langle \mathcal{I}, \prec, \mathcal{D}, \lambda, u\rangle$ *consists of a set of instants* $\mathcal{I}$*, which owns a quasi-order relation* $\prec$*, a set of labels for the instants* $\mathcal{D}$*, a labeling function* λ*, and a unit* u *for the clock ticks. A finite clock has a finite number of ticks. If no ticks are left, the clock is empty. A clock* $c_1 = \langle \mathcal{I}_1, \prec_1, \mathcal{D}_1, \lambda_1, u\rangle$ *is a* subclock *of another clock* $c_2 = \langle \mathcal{I}_2, \prec_2, \mathcal{D}_2, \lambda_2, u\rangle$*, if* $\mathcal{I}_1 \subseteq \mathcal{I}_2$.

Example 1. *Consider a sensor which can perform measurements at certain times. The time steps to perform a measurement can be described using the clock* `sensor1`*. They form a set of instants of the clock which are illustrated by dots at the time line shown in Fig. 1. Every instant represents a clock tick. These ticks represent time steps in which a measurement may be conducted. The order of the instants on the line is specified by* $\prec_{sensor1}$*. Finally, the time steps in which a measurement does actually take place are labeled by a corresponding ID (in Fig. 1, denoted below the dots representing the instants).*

In general, clocks can be logical or chronometric [2]. Logical clocks can refer to discrete events like processor cycles or sensor data, while chronometric clocks refer to physical time and can also be dense. In this contribution, we consider discrete clocks. From the clocks, a time structure can be derived [14]:

Definition 2. *A* time structure *is a pair* $\langle \mathcal{C}, \preccurlyeq\rangle$*, where* $\mathcal{C}$ *is a set of clocks and* $\preccurlyeq$ *is a binary relation on* $\cup_{c\in\mathcal{C}}\mathcal{I}_c$ *named precedence (one clock tick takes place before or coincidentally with the other). From* $\preccurlyeq$*, some further relations can be derived to specify instant behavior.*

```
1  ClockConstraintSystem sensors {
2   Clock minClock;
3   Clock sensor1;
4   Clock sensor2;
5   Clock echo is sensor1 delayedBy 1;
6
7   sensor2 # echo;
8   sensor1 isPeriodicOn minClock period 1.0;
9   sensor2 isPeriodicOn minClock period 1.0;
10 }
```

Fig. 2. A CCSL specification

These instant relations affect the instants to which they are referring to, but not the rest of the instants of the clock. In contrast, if instant relations are defined for all instants of the clock, they become clock relations (or clock definitions, in terms of CCSL). These clock relations constrain the complete behavior of the clocks. An illustration of clock relations is given in the following example. A complete list of clock constraints can be found e. g. in [2].

Example 2. *Consider a system with two sensors triggered by the clocks* `sensor1` *and* `sensor2`*. Both sensors have to reply in every second step with respect to a third (minimal) clock* `minClock`*. From its second reply on,* `sensor1` *causes an echo interfering with the signal of* `sensor2`*. Hence, this echo and the reply of* `sensor2` *are not allowed to occur coincidentally. This is described in the CCSL specification in Fig. 2.*

At the beginning, the clocks are defined (lines 1–5). Clock `echo` *in line 5 is defined as a subclock of* `sensor1`*. Both tick together, but* `echo` *starts one tick after* `sensor1`*. Afterwards, the clock relations are applied. Line 7 states the exclusion of* `echo` *and* `sensor2`*, while lines 8 and 9 define the periodicity of* `sensor1` *and* `sensor2` *on* `minClock`*.*

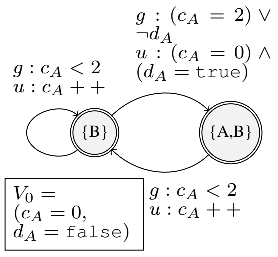
III. Generic Representation of CCSL Constraints

Description means as described above allow for a very precise specification of the timing behavior for a system to be realized. The resulting formal descriptions can already be utilized to automatically perform design tasks such as consistency checking, code generation, or verification. For this purpose, a variety of solutions have been proposed in the recent past [3], [4], [5], [6], [7]. A typical step involved in most of these approaches consists of transforming the given specification or a part thereof into a corresponding representation such as Promela, SystemC, etc. In this work, we are aiming for representing the non-functional timing constraints (provided in CCSL) in terms of a more generic representation which is applicable to existing design approaches for the above mentioned tasks but also can easily be integrated to solutions only addressing functional behavior thus far.

The idea of our representation is to model the timing behavior by means of so-called ticking sets. A *ticking set* describes all clocks $c \in \mathcal{C}$ which can tick in a certain time step. Clocks that are not included in a ticking set will not tick in the respective time step. Relying on such a description, the entire timing behavior can be represented as a sequence of ticking sets: Each ticking set constitutes a *state*; *transitions* allow for moving from one state to another in accordance to

```
A isPeriodicOn B
  period 2.0;
```

(a) CCSL constraints



$g : (c_A = 2) \vee \neg d_A$
$u : (c_A = 0) \wedge (d_A = \texttt{true})$
$g : c_A < 2$
$u : c_A{+}{+}$
{B}
{A,B}
$V_0 = (c_A = 0, d_A = \texttt{false})$
$g : c_A < 2$
$u : c_A{+}{+}$

(b) Corresponding automaton

Fig. 3. Generic representation

the constraints originally provided in CCSL. For this purpose, transitions may have *guard conditions* over global *variables* (such as counters for periodic behaviors) which state whether a transition may be taken or not. The values of these global variables may be changed during a transition by means of *update functions*.

More precisely, the given CCSL constraints shall be represented in terms of a transition relation given by an automaton A defined as follows.

Definition 3. *Let $\mathcal{C}$ be a set of clocks given by a CCSL specification. Furthermore, let $\mathcal{V}$ be a set of global variables used to store additional information (e. g. counters) and V an assignment of them. In order to evaluate conditions derived from the CCSL constraints, a guard function g is applied which maps the current assignment V to either* `true` *or* `false`. *Finally, let u be an update function which manipulates the values of $\mathcal{V}$. Then, the behavior of the clocks according to a given CCSL specification can be represented by an automaton $\mathcal{A} = (\Sigma, S_0, \mathcal{V}, V_0, \delta)$ where*

- *$\Sigma \subseteq \mathcal{P}(\mathcal{C})$ are the states referring to possible ticking sets,*
- *$S_0 \subseteq \Sigma$ are the initial states,*
- *$\mathcal{V}$ is a set of global variables,*
- *V_0 is an initial assignment of $\mathcal{V}$, and*
- *$\delta : \sigma \overset{g,u}{\mapsto} \sigma'$ is a relation representing all transitions from a source state $\sigma \in \Sigma$ to a target state $\sigma' \in \Sigma$ with guard condition g and update function u.*

Obviously a transition can only be used if the guard function evaluates to `true`.

Example 3. *Fig. 3(a) shows a CCSL constraint defining the timing behavior of two clocks* `A` *and* `B` *being periodic. A corresponding transition relation representing this timing behavior is provided by the automaton shown in Fig 3(b). More precisely, from the periodicity of* `A` *on* `B` *one can conclude that* `A` *is a subclock of* `B`*, i. e.* `A` *can never tick alone. Hence, the set Σ of states to consider is composed of the ticking sets {*`B`*} and {*`A,B`*} only. In addition to that, a counter variable c_A and a Boolean variable d_A are utilized in order to buffer the actual period as well as to monitor whether* `A` *has already ticked. Relying on the assignments of these variables, the transitions δ between the respective ticking sets can be conducted as indicated in Fig 3(b).*

IV. Determining the Generic Representation

While the generic representation as introduced in the previous section allows for manifold applications, a structured approach is required to automatically determine this automaton from given CCSL constraints. This section outlines a possible scheme for this purpose. A discussion of the benefits and drawbacks as well as an exemplary evaluation of the proposed approach are later provided in Section V.

A. Initializing Automaton

First, an initial structure for the automaton is created, i. e. an initial set of states which may have to be considered is generated. A clock $c \in \mathcal{C}$ may either tick solely or in conjunction with one or more other clocks. Hence, in the worst case, for each possible subset of clocks, an own state may be required. Additionally, we assume that, in each state, at least one clock is supposed to tick, i .e. there is no "empty" state. Overall, this leads to an initial set Σ of states formed over the power set of all the clocks in $\mathcal{C}$:

$$\Sigma \leftarrow \mathcal{P}(\mathcal{C}) \setminus \{\emptyset\}.$$

In a similar fashion, we initially assume that an arbitrary clocking behavior is allowed and, thus, a transition from each ticking state to any other ticking state is possible. Guard conditions and update functions are initially assumed to always evaluate to `true` and to employ the identity, respectively:

$$\delta \leftarrow \{\sigma \overset{g,u}{\mapsto} \sigma' \mid \sigma, \sigma' \in \Sigma, g(V) = \texttt{true}, u = id_V\}$$

Hence, the automaton is initialized as a complete graph over the power set of all clocks in $\mathcal{C}$.

Example 4. *In the remainder of this section, the necessary steps are illustrated by means of the CCSL constraints given in Fig. 2. This specification of timing behavior is composed of the clocks $\mathcal{C} = \{\texttt{sensor1}, \texttt{sensor2}, \texttt{minClock}, \texttt{echo}\}$, leading to a set Σ composed of 15 states for the desired automaton. Each state is connected to all of the remaining states eventually leading to the initial structure as shown Fig. 4(a). Note that, due to space restrictions, the names of the clocks have been shorted, i. e.* `sensor1`*,* `sensor2`*,* `minClock`*, and* `echo` *have been abbreviated by* `s1`*,* `s2`*,* `mC`*, and* `e`*, respectively.*

B. Applying Constraints

Once the initial structure of the automaton is generated, the constraints are applied to it. For every CCSL constraint, it is checked whether it states a subclock relation (denoted by $\subseteq$) between two clocks. For all clocks $c_1, c_2 \in \mathcal{C}$ with $c_1 \subseteq c_2$, all states containing the subclock but not the superclock are removed. This means, the set of states is adjusted for each $c_1 \subseteq c_2$ as follows:

$$\Sigma \leftarrow \Sigma \setminus \{s \in \Sigma \mid c_1 \in s \wedge c_2 \notin s\}$$

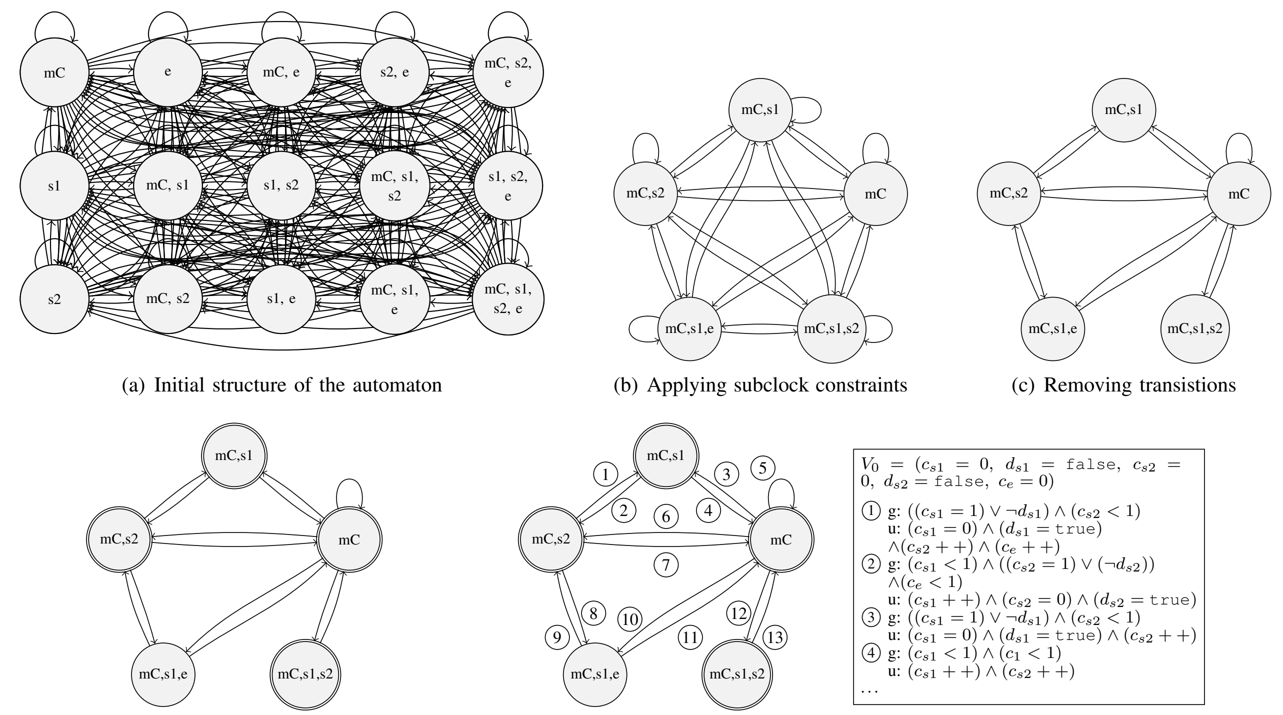
Furthermore, states containing two clocks excluding each other (denoted by $\#$) are removed. Hence, for all clocks $c_1, c_2 \in \mathcal{C}$ with $c_1 \# c_2$ the set of states is adjusted as follows:

$$\Sigma \leftarrow \Sigma \setminus \{s \in \Sigma \mid c_1 \in s \wedge c_2 \in s\}$$

Obviously, the ingoing and outgoing transitions of all dropped states are removed as well.

For sake of clarity, a precise definition e. g. of the domain of a variable $v \in \mathcal{V}$ is omitted here, but will be provided later in Section IV.

Note that, in many cases, the number of states to be considered can already be restricted at the very beginning. As an example, it is already obvious from the CCSL constraints in Fig. 3(a) and discussed in Example 3 that only two states are required. However, in order to keep the following descriptions generic, such cases are not explicitly discussed in the following.



(a) Initial structure of the automaton

(b) Applying subclock constraints

(c) Removing transistions

(d) Determining initial states

$V_0 = (c_{s1} = 0,\ d_{s1} = \texttt{false},\ c_{s2} = 0,\ d_{s2} = \texttt{false},\ c_e = 0)$

① g: $((c_{s1} = 1) \vee \neg d_{s1}) \wedge (c_{s2} < 1)$
u: $(c_{s1} = 0) \wedge (d_{s1} = \texttt{true})$
$\wedge (c_{s2} + +) \wedge (c_e + +)$
② g: $(c_{s1} < 1) \wedge ((c_{s2} = 1) \vee (\neg d_{s2}))$
$\wedge (c_e < 1)$
u: $(c_{s1} + +) \wedge (c_{s2} = 0) \wedge (d_{s2} = \texttt{true})$
③ g: $((c_{s1} = 1) \vee \neg d_{s1}) \wedge (c_{s2} < 1)$
u: $(c_{s1} = 0) \wedge (d_{s1} = \texttt{true}) \wedge (c_{s2} + +)$
④ g: $(c_{s1} < 1) \wedge (c_1 < 1)$
u: $(c_{s1} + +) \wedge (c_{s2} + +)$
…

(e) Refining guard conditions and update functions

Fig. 4. Determining the generic representation

Example 5. *In the considered example, the constraints* `isPeriodicOn` *and* `delayedBy` *indicate subclock relations (see lines 5, 8, and 9 in Fig. 2), i. e. the periodic clocks* `sensor1` *as well as* `sensor2` *are both subclocks of* `minClock` *and* `echo` *is a subclock of* `sensor1`*. Hence, all states containing* `sensor1` *or* `sensor2` *but not* `minClock` *as well as all states containing* `echo` *but not* `sensor1` *are removed. Moreover, the clocks* `echo` *and* `sensor2` *exclude each other (see line 7 in Fig. 3(a)), so that all states containing both are removed as well. The resulting automaton is shown in Fig. 4(b).*

At this stage, some more transitions can be removed. In fact, certain CCSL constraints may reveal that clocks cannot tick directly after each other. Thus, transitions between states containing these clocks can be removed. Although this can also be handled later when guard conditions are refined (which would never evaluate to true for such transitions), dropping them before simplifies the remaining steps significantly.

Example 6. *Consider the* `isPeriodicOn`*-constraints of the example (see lines 8 and 9 in Fig. 2). Since the periods are greater than 0, it can be concluded that there can never be a valid transition between two states containing the respective subclock. Hence, all transitions from states containing* `sensor1` *to states containing* `sensor1` *are removed. The same happens with transitions from/to states containing* `sensor2`*. The resulting automaton is shown in Fig. 4(c).*

Next, the initial states are determined. At the beginning, all states are assumed to be initial states. However, all clocks which are dependent from other clocks (i. e. have to wait for other clocks) obviously cannot initiate the timing behavior. Hence, whenever a CCSL constraint states that a clock $c \in \mathcal{C}$ is dependent on another clock $c' \in \mathcal{C}$ (e. g. clocks with an offset have to wait for a triggering clock tick), all states including c are *not* considered an initial state. Formally, the initial states are defined by

$$S_0 \leftarrow \Sigma \setminus \{\sigma \in \Sigma \mid \exists\, c \in \sigma : c \text{ depends on } c' \in \mathcal{C} \setminus \{c\}\}$$

Example 7. *Thus far, all five states left in the automaton and shown in Fig. 4(c) are assumed to be initial states. However, there exists a dependency between two clocks:* `echo` *has to wait until* `sensor1` *has ticked one time. As a consequence, all states including the clock* `echo` *cannot initiate the timing behavior and, hence, they are not considered to be initial states. Fig. 4(d) shows the resulting automaton.*

Finally, the guard conditions g and the update functions u for the transitions are refined with respect to the CCSL constraints. For this purpose, the set $\mathcal{V}$ of global variables is initialized with

- natural numbers $\mathbb{N}_0$ which are applied as counters e. g. to control periodicity or delays as well as
- Boolean variables $\mathbb{B}$ which are applied to monitor whether a clock has already ticked or not.

For all variables, the initial assignment is added to V_0. Then, for all transitions we refine g and u depending on the respective constraints. For this purpose, a rather direct mapping as illustrated in the following example is applied.

Example 8. *The guard conditions and update functions for the transitions which remained in the automaton shown in Fig. 4(d) have to be refined. While the excluding constraint (line 7 in Fig. 2) has already been handled above (by excluding the corresponding states from the automaton at all), this particularly requires the consideration of the* `isPeriodicOn` *and* `delayedBy` *constraints (see lines 5, 8, and 9 in Fig. 2).*

These constraints obviously require a counter variable $c_{clock} \in \mathbb{N}_0$ (in order to check whether the period or the delay has been completed) and a Boolean variable d_{clock} (in order to store whether the restricted clocks have ticked before) for each of the restricted clocks `clock`*. Note that, in the example, a* `clock` *can either be* `s1`*,* `s2`*, or* `e`*, while the latter just needs a counter and no Boolean variable. Then, states $\sigma \in \Sigma$ including* `clock` *(i. e. with* `clock` *$\in \sigma$) may only be entered either (1) if the period/delay (stored in c_{clock}) satisfies the respective CCSL constraint or (2) if* `clock` *has simply not ticked yet (stored in d_{clock}). In a similar fashion, entering a state $\sigma \in \Sigma$ sets the existing d_{clock}-variables for all* `clock` *$\in \sigma$ to* `true` *(since all* `clock` *$\in \sigma$ are ticking in the following step) and leaving such states updates the corresponding c_{clock}-variables. For the latter, this means either resetting the counter (if the corresponding clock just ticked in σ) or increasing it by one (if the corresponding reference clock ticked).*

Overall, this leads to a revision of the guard conditions and update functions as shown in Fig. 4(e).

Following the scheme described above, almost all CCSL constraints can automatically be mapped into the generic representation. An exception is uncontrollable behavior, e. g. clocks ticking according to external sensor input. However, if the automaton satisfies certain characteristics these constraints are automatically satisfied as well. Hence, supporting uncontrollable behavior boils down to a particular verification task which can e. g. be conducted by established verification methods as discussed in the next section.

V. Discussion & Application

Using the method described above, timing behavior provided in CCSL is generically represented by means of a transition relation. Now, this can be used for various purposes for which only very problem-specific design tools were available thus far. In this section, we discuss possible application areas of the proposed methodology and compare them to related work. Afterwards, the feasibility of the approach is exemplary demonstrated by means of the running example.

A. Application and Comparison to Related Work

An important issue after the specification of timing constraints is to check e. g. (1) whether the resulting CCSL constraints are consistent, i. e. do not contradict each other and thus allow for an execution of the clocks without deadlocks, as well as (2) whether they indeed have been specified as intended. To conduct these checks, a few approaches have previously been proposed, cf. [3], [4], [5], [6]. Here, each constraint is represented by a single automaton. Moreover, they represent ticking sets in terms of transitions rather than states. For some constraints, this leads to an infinite number of states. To deal with that, the authors of [3], [4], [5], [6] restrict the number of transitions.

Our solution approaches this problem from a different direction which allows for an easier consideration of many verification tasks. We rely on a *finite* representation from which serious design flaws can directly be detected. For example, if CCSL constraints result in an automaton composed of states with no outgoing transitions (or outgoing transitions which will never satisfy the guard conditions), a typical deadlock scenario is identified.

TABLE I
CHARACTERISTICS OF THE RESULTING AUTOMATA

INSTANCE	STATES (Σ)	TRANS.	VAR.
sensors4	5	13	5
sensors6	20	117	9
sensors8	80	1053	13

In addition to that, well-known and powerful methods for (bounded) model checking (such as e. g. [8], [9], [10], [11]) can directly be applied on the obtained transition relation in order to check more sophisticated properties. Moreover, the obtained generic representation obtained by our approach can directly be combined to a transition relation representing the functional behavior of a system (obtained e. g. by approaches as proposed in [10], [11]). In doing so, non-functional timing constraints can be considered together with functional specifications. Particular for generic verification frameworks as envisioned e.g. in [13], [15], [16], this is an important benefit.

Another important use case for CCSL descriptions is code generation, i. e. the derivation of a proper implementation of the specified timing behavior. A practical approach for this purpose has recently been presented in [7]. However, this solution relies on a rather heavy data-structure as well a complex analysis scheme including e. g. a categorization, several lists, etc. In contrast, the generic representation of timing constraints proposed in this work provides an easy and obvious basis for code generation. In fact, once an automaton has been found to be correct, it can directly be translated into an implementation by (1) mapping all states to an encoding in terms of state signals (which are connected to the clocks) and (2) mapping all transitions to if-statements ensuring the correct assignment to the state signals.

On the contrary, the proposed approach obviously suffers from the possibly exponential size of the automaton. In the generic approach as described in Section IV, a power set construction is conducted which represents the bottleneck of the solution. However, this worst case behavior can be avoided in many cases. For example, when CCSL constraints as discussed before in Fig. 2 are considered, it can already been derived that only five states (rather than $2^4 - 1 = 15$) are required. Determining more sophisticated bounds for all possible cases remains an open problem for future work, but certainly provides further room for improvement.

B. Exemplary Evaluation

In order to evaluate the applicability of the proposed approach, the running example considered above (i. e. the CCSL constraints from Fig. 2) was subject to more detailed investigations. More precisely, we applied the resulting generic description to selected code generation and verification tasks. In order to evaluate the scalability of the generic representation, we additionally did not only consider this example with four clocks (denoted by *sensors4*), but also derivations including six and eight clocks (denoted by *sensors6* and *sensors8*, respectively). To this end, corresponding CCSL constraints have been added for the newly introduced clocks.

Applying the approach proposed in Section IV to the resulting CCSL instances yielded generic representations with characteristics as summarized in Table I. The columns denote the name of the instance (INSTANCE) as well as the number of states (STATES), the number of transitions (TRANS.), and the number of variables (VAR.). All automata have been derived in negligible run-time, i. e. within a few seconds.

TABLE II
VERIFICATION RESULTS (CONSISTENCY)

INSTANCE	#TRANS. STEPS	RUN-TIME	RES.
sensors4	2	1.7	✓
	50	3.5	✓
	100	5.9	✓
sensors6	2	3.3	✓
	50	24.0	✓
	100	73.0	✓
sensors8	2	48.1	✓
	50	825.7	✓
	100	2583.3	✓

TABLE III
VERIFICATION RESULTS (DEADLOCK)

INSTANCE	#TRANS. STEPS	RUN-TIME	RES.
sensors4	3	2.3	✗
	10	3.9	✗
	30	11.7	✗
sensors6	3	7.1	✗
	10	19.9	✗
	30	122.5	✗
sensors8	3	225.7	✗
	10	928.0	✗
	30	8138.2	✗

Afterwards, typical design tasks have been conducted in which the generic representation has explicitly been utilized. Since code generation can be performed by a rather simple mapping scheme (as discussed above), our evaluations on verification are summarized in more detail in the following.

More precisely, we utilized the resulting generic description in order to automatically prove whether

1) the originally given CCSL description is *consistent*, i. e. allows for a consistent execution at least for a given number of transition steps,
and
2) the originally given CCSL description is *deadlock-free*, i. e. no path from an initial state to another state exists from which no further ticks can be executed anymore.

To this end, the generic description together with the respective verification task has simply been passed to a model checker (in this case, the solution introduced in [10] which has been originally proposed for the verification of UML/OCL models).

Table II and Table III summarize the obtained results for the consistency- and the deadlock-checks, respectively. Again, the first column denotes the name of the instance (INSTANCE), while afterwards the number of considered transition steps (#TRANS. STEPS), the overall run-time (in CPU seconds and obtained on an Intel(R) Core(TM) i5-3320M machine with 2.60 GHz and 8 GB of memory; RUN-TIME), and the result (RES.) is provided. For the last column, a ✓ denotes that consistency/deadlock-freeness has been proven; otherwise, ✗ denotes that an corresponding flaw has been found.

The results confirm the applicability of the proposed approach for verification purposes. In fact, the considered tasks could have been addressed in acceptable run-time and, for a given CCSL constraint, consistency could have been proven by utilizing an existing model checker rather than developing a specific CCSL-based solution. More importantly, the proposed solution even helped unveiling a severe design flaw which was hidden the entire time in the considered running example: The execution of the state sequence $\{\texttt{mc}\} \rightarrow \{\texttt{mC, s1, s2}\} \rightarrow \{\texttt{mC}\}$ leads to a deadlock. This is because the periodicity forces `s1` and `s2` to tick in the next step (`mC` ticks always), but if `s1` ticks, `e` has to tick as well (because of the `delayedBy` constraint). The exclusion between `s2` and `e` eventually leads to a contradiction and, hence, no further transition can be taken at that point. While not obvious at a first glance, this flaw can be detected using the proposed generic representation (as indicated by column RES. in Table III). Again, all these results have been achieved without explicitly developing a specific CCSL-checker but entirely relying on existing tools.

VI. CONCLUSIONS AND FUTURE WORK

This work introduced a generic representation of MARTE CCSL constraints which can easily be utilized for many purposes such as verification, code generation, etc. Besides the definition of the representation, also a procedure to derive it from the given CCSL constraints is provided. A discussion with respect to related work as well as an exemplary evaluation showed the promise of the proposed generic representation. Future work focuses on improving the size of the automaton during the generation process as discussed in Section V-A as well as more detailed evaluations of the proposed methodology in further application areas.

ACKNOWLEDGMENTS

This work has been partially supported by the Graduate School SyDe, funded by the German Excellence Initiative within the University of Bremen's institutional strategy and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1 and a research project under grant no. WI 3401/5-1.

REFERENCES

[1] Object Management Group, *OMG Systems Modeling Language (OMG SysML^{TM})*. Object Management Group, 2012.
[2] ——, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. Object Management Group, 2011.
[3] F. Mallet and L. Yin, *Correct Transformation from CCSL to Promela for verification*. Institut National de Recherche en Informatique et en Automatique, 2012.
[4] J. Suryadevara and L. Yin, "Timed Automata Modeling of CCSL Constraints," in *International Workshop on Formal Techniques for Safety-Critical Systems*, 2012, pp. 152–157.
[5] L. Yin, F. Mallet, and J. Liu, "Verification of MARTE/CCSL Time Requirements in Promela/SPIN," in *International Conference on Engineering of Complex Computer Systems*, 2011, pp. 65–74.
[6] C. André, F. Mallet, and J. DeAntoni, "VHDL Observers for Clock Constraint Checking," in *International Symposium on Industrial Embedded Systems*, 2010, pp. 98–107.
[7] J. Peters, R. Wille, and R. Drechsler, "Generating SystemC Implementations for Clock Constraints Specified in UML/MARTE CCSL," in *International Conference on Engineering of Complex Computer Systems*, 2014, pp. 116–125.
[8] X. Li, Z. Liu, and J. He, "Consistency checking of UML requirements," in *International Conference on Engineering of Complex Computer Systems*, 2005, pp. 411–420.
[9] J. Cabot, R. Clarisó, and D. Riera, "Verifying UML/OCL Operation Contracts," in *Integrated Formal Methods*, 2009, pp. 40–55.
[10] M. Soeken, R. Wille, and R. Drechsler, "Verifying Dynamic Aspects of UML Models," in *Design, Automation and Test in Europe Conference*, 2011, pp. 1–6.
[11] M. Gogolla, L. Hamann, F. Hilken, M. Kuhlmann, and R. B. France, "From application models to filmstrip models: An approach to automatic validation of model dynamics," in *Modellierung*, 2014, pp. 273–288.
[12] E. Ebeid, D. Quaglia, and F. Fummi, "Generation of SystemC/TLM code from UML/MARTE sequence diagrams for verification," in *Symposium on Design and Diagnostics of Electronic Circuits & Systems*, 2012, pp. 187–190.
[13] R. Wille, M. Gogolla, M. Soeken, M. Kuhlmann, and R. Drechsler, "Towards a generic verification methodology for system models," in *Design, Automation and Test in Europe Conference*, 2013, pp. 1193–1196.
[14] C. André and F. Mallet, *Clock Constraints in UML/MARTE CCSL*. Institut National de Recherche en Informatique et en Automatique, 2008.
[15] C. Hilken, J. Peleska, and R. Wille., "A unified formulation of behavioral semantics for SysML models," in *International Conference on Model-Driven Engineering and Software Development*, 2015.
[16] C. Hilken, J. Seiter, R. Wille, U. Kühne, and R. Drechsler, "Verifying consistency between activity diagrams and their corresponding OCL contracts," in *Forum on Specification and Design Languages*, 2014.