

Verifying MARTE/CCSL Mode Behaviors using UPPAAL

Jagadish Suryadevara¹, Cristina Seceleanu¹, Frédéric Mallet² and Paul Pettersson¹

¹ Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden

² Aoste Team-project INRIA/I3S, Sophia-Antipolis, France

jagadish.suryadevara@mdh.se, cristina.seceleanu@mdh.se,
frederic.mallet@unice.fr, paul.pettersson@mdh.se

Abstract. In the development of safety-critical embedded systems, the ability to formally analyze system behavior models, based on timing and causality, helps the designer to get insight into the systems overall timing behavior. To support the design and analysis of real-time embedded systems, the UML modeling profile MARTE provides CCSL – a time model and a clock constraint specification language. CCSL is an expressive language that supports specification of both logical and chronometric constraints for MARTE models. On the other hand, semantic frameworks such as timed automata provide verification support for real-time systems. To address the challenge of verifying CCSL-based behavior models, in this paper, we propose a technique for transforming MARTE/CCSL mode behaviors into Timed Automata for model-checking using the UPPAAL tool. This enables verification of both logical and chronometric properties of the system, which has not been possible before. We demonstrate the proposed transformation and verification approach using two relevant examples of real-time embedded systems.

1 Introduction

The increasing complexity and safety-criticality of real-time embedded systems in domains such as automotive and avionics, stress the need for applying rigorous analysis techniques during system development in order to ensure predictability [8]. To meet this need, the standard modeling language UML (The Unified Modeling Language) [14] provides a domain-specific modeling profile called MARTE (Modeling and Analysis of Real-Time and Embedded systems) [15]. Besides modeling support for *performance* and *schedulability* analysis, MARTE includes CCSL – a time model and a clock constraint specification language, for describing both logical and physical (chronometric) clock constraints [3]. On the other hand, semantic frameworks such as timed automata provide modeling and verification support for real-time systems [1,7,11,2], which CCSL - based models could benefit from. An important feature of CCSL is that it can be used for expressing/specifying both synchronous and asynchronous constraints, based on the *coincidence* and *precedence* relationships between clock instants. However, the expressiveness of CCSL poses challenges with respect to providing rigorous analysis support, like exhaustive verification, to its specifications. The focus of our work, in this paper, is to address these challenges and provide a model-checking based

verification support for MARTE/CCSL behavior models, by transforming them into the timed automata (TA) framework.

MARTE Statemachine models, called ModeBehaviors, can be used to specify the *mode*-based behavior of a system. In this view, a *mode* represents an *operational segment* within the system execution that is characterized by a *configuration* of system entities. For instance, during ‘TakeOff’, ‘Flying’ and ‘Landing’ modes of an aircraft, different parts of the control system may be active in different modes.

In this paper, we propose to constrain MARTE mode behaviors with CCSL specifications, using the underlying MARTE time model. As we show, this facilitates precise specification of logical (of synchronous and asynchronous nature) as well as physical (chronometric) time properties of a system within a mode. Next, as a main contribution, we present a technique to transform MARTE/CCSL mode behaviors into timed automata [1,7] to enable model-checking based analysis. The transformation is based on the synchronized product of the state-based representations of the CCSL semantics [4,12]. This proves to be non-trivial due to the expressiveness of CCSL constraints and the differing semantic domain of TA framework.

In brief, in this paper, we make the following contributions:

- We provide a mapping strategy to transform CCSL-extended MARTE mode behaviors into timed automata, and verify logical and chronometric properties using the UPPAAL model-checking tool [11].
- We propose novel techniques to address the limitations of mapping synchronous and chronometric semantics of CCSL into timed automata.
- We demonstrate the proposed modeling and verification approach using simplified versions of two representative examples of safety-critical embedded control systems, namely, a *temperature control system* and an *anti-lock braking system*.

The rest of the paper is organized as follows. In Section 2, we introduce example embedded systems and present the corresponding mode behavior specifications. In Section 3, we present an overview of CCSL, followed by the CCSL extended mode behaviors for the example systems. In Section 4, we present the proposed transformation technique for CCSL-extended mode behavior specifications, and in Section 5, we discuss verification results based on the transformed timed automata models of the example systems. The related work is discussed in Section 6. Finally, we conclude the paper in Section 7, with a discussion of future work.

2 Example Systems and Mode-behavior Specifications

In this section, we present the mode behavior specifications of the example embedded systems used in this paper. We have chosen two simple but representative systems, which represent different kinds of functional and timing aspects commonly found in embedded systems.

MARTE Notations and Stereotypes. In MARTE, the stereotype *ModeBehavior* extends the UML Statemachine notation with stereotypes *Mode*, which extends State, and

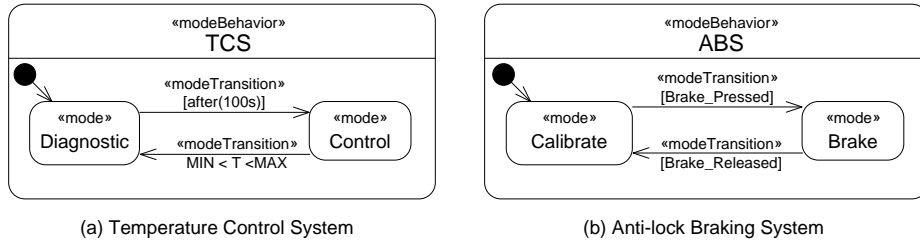


Fig. 1. UML/MARTE mode behavior specifications.

ModeTransition that extends *Transition* (Fig. 1). A *ModeBehavior* specifies a set of mutually exclusive modes, that is, only one mode can be active at a given instant. A *mode* represents an operational fragment of the system, also called *configuration*, meaning the set of system entities that are active during the mode instance. The dynamics of mode switching, either time or event triggered, is specified by connecting *modes* by means of *ModeTransitions*. Transitions are enabled as a response to an event occurrence, that is, the activation condition triggering the mode switching.

2.1 Example1: A Temperature Control System (TCS)

We consider a simplified version of a temperature control system that regulates the temperature inside a nuclear reactor core, by using thermal-controlling rods. The rods are inserted into the core of the reactor when the temperature reaches a given upper limit, denoted by constant *MAX*, causing the temperature to gradually reduce (as neutrons in the reactor are absorbed by the control rods). Similarly, the control rods are removed when the temperature in the reactor falls below *MIN*. TCS operates in two *modes*.

In Diagnostic mode, the following *actions* are triggered that execute the corresponding *behaviors*³: *Diagnostics* examines the current status of the control rods, *Reconfig* replaces the ineffective control rods if any, and *StatusUpdate* updates the status of a rod configuration in the reactor. In Control mode, the system triggers three actions; *PeriodicSense* senses the temperature in the reactor, *InsertRod* inserts a control rod, and *RemoveRod* removes a rod from the reactor.

The TCS mode behavior is presented in Fig. 1. After 100 s in Diagnostic mode, the system changes to Control mode. However, the mode-change from Control to Diagnostic is triggered by an event occurrence, indicating the sensed temperature in the reactor is within the specified limits. The following properties specify the functional and timing aspects for TCS:

- TCS1 : *Diagnostics* is always followed by *Reconfig*.
- TCS2 : The behavior of *Reconfig* is ‘extended’ by *StatusUpdate*, only when there is a change in the control rod configuration.
- TCS3 : *PeriodicSense* executes periodically with a period of 10 s.
- TCS4 : *PeriodicSense* is followed by *InsertRod* or *RemoveRod* but not both.
- TCS5 : At most two rods can be used in sequence, for cooling the reactor core.

³ By *behavior*, we refer to primitive functionality often implemented as a single piece of code. We assume instantaneous execution of a behavior, if not specified otherwise.

2.2 Example2: An Anti-lock Braking System (ABS)

ABS is a control unit in a car that ensures the stability of the vehicle during drive and extreme brake situations. It functions in two operational modes: Calibrate and Brake. The default mode is Calibrate. In this mode, the system maintains the required speed equally on all the four wheels, by calibrating and adjusting the current speeds on individual wheels. In Brake mode, the ABS ensures lock-free application of brake pressure on all the wheels, enforcing the car's stability, in particular on slippery surfaces.

In the Calibrate mode, the ABS triggers two actions: `SenseSpeed` periodically senses the current wheel speed values, and `Calibrate` estimates the speed to be adjusted on each individual wheel with respect to the required speed. In the Brake mode, ABS triggers three actions: `SenseBrake` that receives the current brake torque value, `BrakeControl` that determines the brake pressure to be applied, and `BrakeWheel` that applies required brake pressure with anti-lock braking to individual wheels.

The ABS *mode*-behavior is shown in Fig. 1. The mode changes are caused by events *Brake_Pressed* and *Brake_Released*. The following properties specify the functional and timing constraints in ABS.

- ABS1 : `SenseSpeed` is always followed by `Calibrate`.
- ABS2 : `SenseSpeed` is periodic with a period of 100 ms.
- ABS3 : `Calibrate` completes within 10 ms after `SenseSpeed`.
- ABS4 : `SenseBrake` is always followed by `BrakeControl`.
- ABS5 : `BrakeControl` is always followed by `BrakeWheel`.
- ABS6 : `SenseBrake` is periodic with a period of 10 ms.
- ABS7 : `BrakeWheel` completes within 1 ms after `SenseBrake`.

In the next section, we extend the mode behavior specifications of TCS and ABS, using CCSL constraints for the specification of logical and chronometric properties described above.

3 CCSL

UML/MARTE provides modeling support to capture structural as well as functional and extra-functional behavioral aspects of systems. The Clock Constraint Specification Language (CCSL [4]), initially specified in an annex of MARTE, provides an expressive set of constructs to specify causality (of both synchronous and asynchronous nature) as well as chronological and timed properties of system models, and it has been used in various subdomains. The CCSL is formally defined and CCSL specifications can be executed at the model level [9].

3.1 CCSL Constraints

CCSL is a declarative language that specifies constraints imposed on the *clocks* (activation conditions) of a model. CCSL clocks refer to any repetitive events of the system and should not be confused with UPPAAL clocks. CCSL clocks are defined as an (often infinite) sequence of clock *instants* (event occurrences). If *c* is a CCSL clock,

$c[k]$ denotes its k^{th} instant, for any $k \in \mathbb{N}$. Below, we describe only the constraints used in this paper. A comprehensive description of CCSL constructs can be found in André's work [4].

Synchronous constraints rely on the notion of *coincidence* of clock instants. For example, the clock constraint `a isSubclockOf b`, denoted by $a \boxed{\subset} b$, specifies that each instant of *subclock* a must coincide with exactly one instant of *superclock* b . Other examples of synchronous constraints are `discretizedBy` or `excludes` (denoted $\boxed{\#}$). The latter prevents two clocks from ticking simultaneously. The former discretizes a dense clock to derive discrete chronometric clocks, mostly from *IdealClk*, a perfect dense chronometric clock, predefined in MARTE Time Library, and assumed to follow 'physical time' faithfully (with no jitter).

Asynchronous constraints are based on instant *precedence*, which may appear in a strict (\prec) or a non-strict (\preceq) form. The clock constraint `a isFasterThan b` (denoted by $a \boxed{\prec} b$) specifies that clock a is (non-strictly) faster than clock b , that is for all natural number k , the k^{th} instant of a precedes or is coincident with the k^{th} instant of b ($\forall k \in \mathbb{N}; a[k] \preceq b[k]$). *Alternation* is another example of an asynchronous constraint. It is a form of bounded precedence. The constraint `a alternatesWith b` (denoted by $a \boxed{\sim} b$ or $a \boxed{\prec_1} b$) states that $\forall k \in \mathbb{N}; a[k] \prec b[k] \wedge b[k] \prec a[k+1]$, i.e., an instant of a precedes the corresponding instant of b which in turn precedes the next instant of a .

Mixed constraints combine *coincidence* and *precedence*. The constraint `c = a delayedFor n on b` enforces a delayed coincidence, i.e., imposes c to tick synchronously with the n^{th} tick of b following a tick of a . It is considered as a mixed constraint since a and b are not assumed to be synchronous.

Table 1. CCSL constraints for TCS and ABS systems.

<i>Property</i>	<i>CCSL Constraints</i>
TCS3	Clock $p \boxed{=} \text{IdealClk}$ discretizedBy 10 s
ABS2	Clock $s \boxed{=} \text{IdealClk}$ discretizedBy 0.1s
ABS6	Clock $b \boxed{=} \text{IdealClk}$ discretizedBy 0.01s
TCS2	$s \boxed{\subset} c$
TCS1	$d \boxed{\sim} c$
TCS4	$p \boxed{\sim} (i \cup r) \wedge i \boxed{\#} r$
TCS5	$i \boxed{\prec_2} r$
ABS1	$s \boxed{\sim} l$
ABS4	$r \boxed{\sim} w$
ABS3	$l \boxed{\prec} s$ delayedFor 1 on c_1 where Clock $c_1 = \text{IdealClk}$ discretizedBy 0.01s
ABS7	$w \boxed{\prec} b$ delayedFor 1 on c_2 where Clock $c_2 = \text{IdealClk}$ discretizedBy 0.001s

3.2 CCSL Constraints for TCS and ABS

The functional and timing properties of the TCS and ABS, as CCSL constraints, are given in Table 1. These properties constrain the system behaviors with respect to causality and timing. The constraints are listed in three groups: *synchronous*, *asynchronous*, and *mixed*. The correspondence between the actions in TCS mode behavior and the logical clocks in the CCSL constraints is as follows: Diagnostics: d , Reconfig: c , StatusUpdate: s , PeriodicSense: p , InsertRod: i , and RemoveRod: r . Similarly, for the ABS system, the correspondence between the primitive behaviors and the logical clocks is as follows: SenseSpeed: s , Calibrate: l , SenseBrake: b , BrakeControl: r , and BrakeWheel: w .

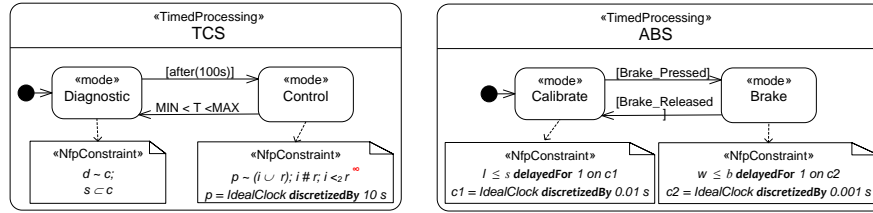


Fig. 2. MARTE/CCSL mode behavior specifications.

In Fig. 2, we present the CCSL-extended mode behavior specifications of TCS and ABS. We use MARTE stereotype ‘TimedProcessing’ for mode behaviors, as we associate modes with CCSL clocks. We also use stereotype ‘NfpConstraint’ to associate CCSL constraints to a mode. However, in this paper, we distinguish between the stateful CCSL-constraints that retain history during complete system ‘runs’ from those that retain history during a *mode* execution. History-enabled CCSL constraints are annotated with symbol ∞ , for instance, the constraint $i \prec_2 r$ for TCS Control mode.

4 MARTE/CCSL Mode Behaviors to Timed Automata

In this section, we propose a mapping strategy to transform MARTE/CCSL mode behaviors, henceforth simply referred to as mode behaviors, into TA, to provide UPPAAL model-checking support. We first present a brief overview of timed automata as used in UPPAAL.

4.1 Timed automata and UPPAAL: An overview

A timed automaton (TA) is a tuple $\langle L, l_0, C, A, E, I \rangle$, where L is a set of *locations*, $l_0 \in L$ is the initial location, C is the set of clocks, A is the set of actions, synchronization actions and the internal τ -action, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard, a set of clocks to be reset, and $I : L \rightarrow B(C)$ assigns clock *invariants* to locations. A location can be marked *urgent* (u) or *committed* (c) to indicate that the time is not allowed to progress in the specified location(s), the latter being a stricter form indicating further that the next transition

can only be taken from the respective committed location only. Also, synchronization between two automata is modeled by *channels* (e.g., $x!$ and $x?$) with rendezvous or broadcast semantics.

UPPAAL extends the timed automata language, originally introduced by Alur and Dill [1], with a number of features such as global and local (bounded) integer variables, arithmetic operations, arrays, and a C-like programming language. The tool consists of three parts: a graphical editor for modeling timed automata, a simulator for trace generation, and a verifier for the verification of a system modeled as a network of timed automata. A subset of CTL (computation tree logic) is used as the input language for the verifier. For further details, we refer the reader to UPPAAL tutorial [11].

4.2 Synchronized Product of CCSL Constraints: An example

A state-based semantics of CCSL operators has been defined [12], using the Labelled Transition Systems (LTS). With this, the combined LTS of composed CCSL operators can be obtained, using the synchronized product of the LTSs [6]. As an example, we present the synchronized product for CCSL constraints in the TCS Diagnostic mode, as shown in Fig. 3. The LTS of the constraint $d \text{ alternatesWith } c$ is presented in Fig. 3.(a). It specifies that only the clock d can tick in state 1, whereas in state 2 only the clock c . Thus, it intuitively specifies the semantics of the operator `alternatesWith` ($d \text{ alternatesWith } c$). An empty transition, denoted by ϵ , represents that no clock ticks, but useful for composing two LTSs. The LTS of $s \text{ synchronizes } c$, as shown in Fig 3.(b), specifies that, in state A, either only c ticks or both s and c tick synchronously (denoted by $\langle s, c \rangle$). The synchronized product of the above described LTS, as shown in Fig 3.(c), considers all possible states and the transitions. For instance, in the state 2A, the only non- ϵ transition in state 2 of the first LTS, combines with either the ϵ, c -transition, or $\langle s, c \rangle$ transition in state A of the second LTS, resulting in the two possible transitions i.e. c , or $\langle c, s \rangle$. Further details on the synchronization products of CCSL constraints, are described by Mallet [12].

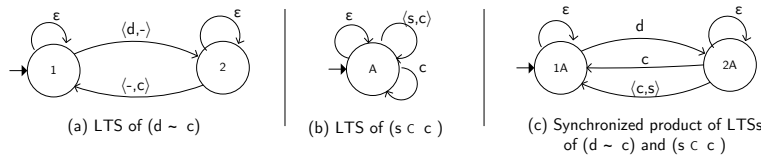


Fig. 3. Synchronized product of LTS-based CCSL semantics.

4.3 Transforming Mode Behaviors into Timed automata

For the transformation of a MARTE/CCSL mode behavior into a timed automaton, several aspects need to be considered, such as, logical clocks, CCSL constraints, logical and chronometric time, modes and mode transitions. The transformation consists mainly of three steps: mapping CCSL constraints of modes into corresponding TA, referred to as LTS-TA, modeling logical and chronometric timing aspects in the transformed TA, and transforming modes and mode transitions. The mapping strategy is summarized in Fig. 4.



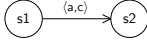
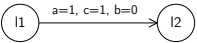
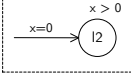
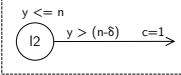
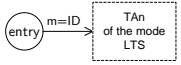
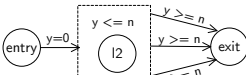
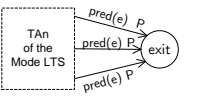
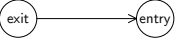

LTS / CCSL / Modes	Timed Automata	Remarks
		A state in a LTS is mapped to a location in the corresponding TAn.
Logical clock 'a'	Boolean variable 'a'	A 'ticking' of the logical clock denoted by the boolean value '1', and non-ticking by '0'.
		A transition with a 'ticking' configuration in a LTS is mapped to a TAn edge with an update action of the boolean variables corresponding to logical clocks that 'tick' synchronously, other boolean variables are set to '0'
Non-deterministic durations of logical clock 'configurations'		On every edge in the LTS-TAn of modes, a global clock variable 'x' is reset, and invariant 'x>0' assigned to all locations.
Logical clock 'c' with chronometric period 'n' ms (i.e. 'c' = IdealClock discretizedBy n ms')		For every location in the LTS-TAn, with outgoing edge containing action 'c=1', add the invariant 'y<=n' to the location and guard 'y > (n-delta)' to the edge, where 'y' is a clock variable, and '0<delta<n' is the minimum jitter necessary to integrate logical steps with chronometric time progress.
CCSL-annotated mode		New urgent-location 'entry'. Edges from 'entry' to initial location of the TAn of the mode LTS. Also, a global variable 'm' assigned the mode identifier.
Time-triggered mode-transition e.g 'after(n ms)'		For every location in the LTS-TAn, add the invariant 'y<=n', and an edge, with guard 'y>=n', to the new 'exit' location (urgent), where 'y' is a clock variable, reset on entry to the mode.
Event-triggered mode-transition e.g 'event e'		For every location in the LTS-TAn, add edge with guard 'pred(e)' to the new 'exit' location (urgent). Add priority channel 'P', to force the transition when 'pred(e)' holds, i.e. the event 'e' occurs,
Mode-change behavior of a mode-transition (m1 to m2)		Add an edge from the corresponding exit location of the transition to the entry location of the mode-TAn of m2.
History-enabled CCSL constraint		New edges from the entry location to all the locations of the corresponding LTS-TAn, and from latter to all the exit-locations of the mode-TAn. $id1()$, $id2()$ are location identifier functions to support history-enabled constraints.

Fig. 4. MARTE/CCSL mode behaviors to timed automata: A mapping strategy.

Logical clocks and CCSL Constraints. For the logical CCSL constraints in modes, we construct their synchronized products, using the LTS-based semantics of the constraints. These LTS are then transformed into TA, following the mapping strategy presented in Fig. 4. States are mapped to locations, transitions become edges in the corresponding TA. Further, the logical clocks are denoted by boolean variables, with 'ticking' configurations modeled as the action updates of the boolean variables for the TA edges corresponding to the LTS transitions.

Logical and Chronometric Time. The transformation provides a basis, to correlate logical semantics of CCSL through chronometric time progress in TA. This is done by extending the LTS-TA of the modes, described above, with timing mechanisms consisting of clock-variables, clock-guards, and invariants in TA. To begin with, every location in the LTS-TAn of a mode, is assigned the invariant $x > 0$, where x is a clock variable which is also reset on every edge in the TAn. This models the non-deterministic

occurrences of the logical clock configurations. As the next step, we map the CCSL constraints that specify chronometric durations for the logical clocks (for some constraints, we need to separate logical and chronometric parts into separate constraints, as explained for ABS mode behavior transformation later in the section). For instance, a CCSL clock c with period ‘ n ms’ is mapped using the invariant $y \leq n$ on all locations with an outgoing edge with action update $c = 1$. Also the edge is assigned the clock guard $y > (n - \delta)$. The value $\delta \ll n$ is necessary to model the integration of logical steps with chronometric time.

Modes and Mode Transitions. Using the timing extended LTS-TA of modes, described above, we obtain the timed automata for modes, by simply adding an `entry`-location and an edge from the new location to the initial location of the corresponding LTS-TAn. A global mode variable m may be updated with the mode identifier value. The mode TA are further extended to enable the mode transitions, as described below.

A mode transition, is either time- or event-triggered, and represents the corresponding mode-change behavior. A mode transition is mapped into a new `exit`-location in the corresponding source mode automaton, and new edges from every location of the mode TAn to the exit location. Additionally, for the time-triggered transition, that is, of the form ‘`after(10ms)`’, we add the invariant $x \leq n$ to all the locations in the mode TAn. Also, the guards of the form $x \geq n$ are assigned to all the edges to the corresponding exit-location. For the event-triggered mode-transition, an event, i.e. ‘ e ’, is mapped by adding the event predicate, denoted by $P(e)$, as guard on all edges to the corresponding exit-location. To force the transition in case of event occurrence, we also use an urgent synchronization channel ‘ $p!$ ’. Finally, in both cases, the mode-switch behavior, corresponding to the transition, is modeled by connecting the exit-location of the source mode TAn to the entry-location of the target mode TAn.

Mode history. For instance, the *Control* mode of the TCS (Fig. 2) contains a history-enabled constraint $i \leq_2 r$. This specifies that the clock i (for `InsertRod`) can tick faster than the clock r (for `RemoveRod`) but not by more than two instances. Clearly, the state of the constraint needs to be retained if the mode is exited and re-enabled later. When a mode is transformed into a TAn, we use a history variable h that is updated on all the edges leading to the exit-location. Moreover, edges are added from the entry-location of the mode to all the locations (not just the initial location), with guards based on the variable h . However, to support the history mechanism, we assume transformation functions ‘`id1()`’ and ‘`id2()`’ that return the location information.

In this section, we have presented some techniques to transform MARTE/CCSL mode behaviors into timed automata. A complete formal transformation and related methodology is out of the paper’s scope. However, we demonstrate the proposed mapping strategy using the mode behavior specifications of the example systems, presented earlier in this paper. We will discuss some additional issues in applying the techniques.

4.4 The transformed automaton for the TCS

In Fig. 5, we present the complete TAn model for the CCSL-extended mode behavior (Fig. 2) of the TCS. The *Diagnostic* mode is transformed to a TAn using the synchronized product of constraints $d \sim c$ and $s \leq c$. Similarly, the *Control* mode is

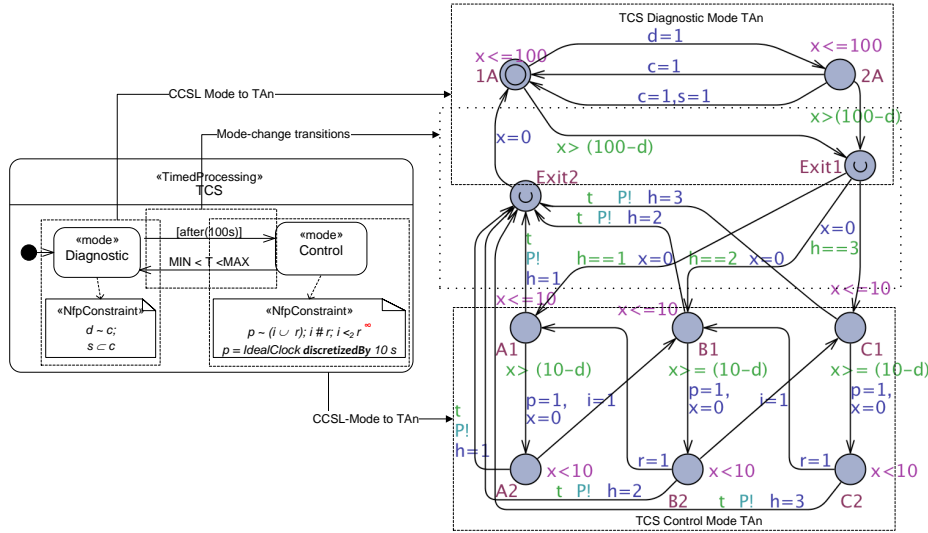


Fig. 5. TCS mode behavior to timed automaton.

transformed using the synchronized product of the constraints $p \approx (i \cup r)$, $i \# r$, and $i \prec_2 r$. These mappings are shown in Fig. 5. For simplicity, the entry-locations of the mode TA are merged with the initial locations in the corresponding LTS-TA, respectively.

Next, we have mapped the mode-transitions that trigger the mode-change behavior, as follows: the time-triggered transition from Diagnostic to Control is mapped using the invariant $x \leq 100$ at the locations of the Diagnostic TAn, and guards $x \geq 100$ for the edges to the exit-location. The mode-transition from Control to Diagnostic is event-triggered, by the predicate denoted by “ r ” (after the required temperature is sensed). Finally, the mode-switchings for the above transitions are modeled by connecting the exit location of the source mode TAn to the initial location of the target mode TAn.

```

1 From CCSL 'alternatesWith' definition:
2
3 left  $\approx$  right :  $\forall i \in \mathbb{N}, p[i] < \text{right}[i] \ \& \ \text{right}[i] < \text{left}[i+1]$  (where  $\mathbb{N}$ , set of
   natural numbers).
4 And,  $p \approx (i \cup r)$  :  $\forall i \in \mathbb{N}^*, p[i] < (i \cup r)[i] \ \& \ (i \cup r)[i] < p[i+1]$ 
5 Given 'p' periodic, i.e. n seconds :  $\forall i \in \mathbb{N}, p[i] = s[n*i - (n-1)]$  where 's' is a
   chronometric clock that counts the seconds.
6
7 For  $n=10, \forall i \in \mathbb{N}, s[10*i - 9] < (i \cup r)[i] < s[10*i + 1]$ 

```

Listing 1.1. Timing invariants derived from CCSL constraints.

The Control mode of the TCS contains a chronometric constraint for the logical clock p (for PeriodicSense). This is mapped to the location invariant $x \leq 10$, and guard $x \geq 10$ for the edges containing the clock ‘ticks’ i.e. $p=1$. However, the other locations also need to be assigned the invariant, due to causality among the CCSL clocks. From the proof given in Listing 1.1, and under the assumption that the physical

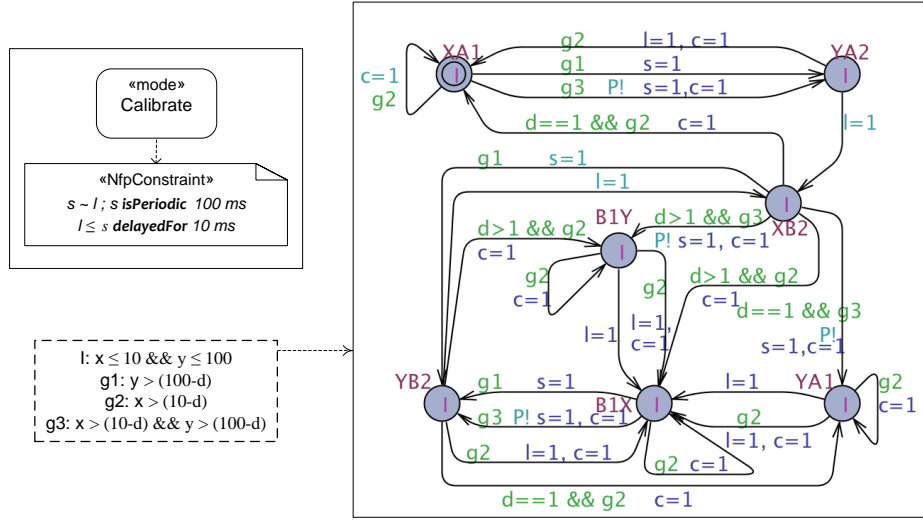


Fig. 6. Timed automaton for ABS Calibrate mode.

time (in TA) is s and $s[1]$ is time 0, we infer $s[10]$ is time 10. For $i=1$, $s[1] < (i \cup r[i] < s[11])$, which gives the interval $(0,10)$. This proves the invariants for the locations.

The Control mode contains the history-enabled CCSL constraint $i \prec_2 r$. The execution state of the constraint, that is, the current location before exiting the TAN, is saved in a history variable h' , when the mode is exited. Based on the saved location identifier, the initial location is chosen, when the mode is re-entered. The history mechanism integrates the expressiveness of CCSL constraints and the mode behavior formalism.

4.5 The transformed automaton for the ABS

For the transformation of the ABS mode behavior, we chose to skip the complete automaton model, and focus only on the transformation of Calibrate mode, given that the CCSL specification of the Brake mode is similar.

Obtaining the synchronized product of CCSL constraints for the Calibrate mode is complex, due to the mixed constraint `delayedFor`. However, we propose a novel technique to address this. We separate the causality and the chronometric aspects for the constraint, using an auxiliary logical clock c , such that the chronometric duration is specified as ‘logical’ ticks of c with additional constraint on c that specifies the actual chronometric duration. This facilitates an easier construction of the synchronized product and also an efficient mapping of the chronometric time to TA. Note that the invariant I (partly) and the guard $g2$ (in the mode automaton of Fig. 6) are due to the chronometric constraint on c (i.e. 10ms), from the mixed CCSL constraint. Also, the invariant I in all locations is due to the other chronometric constraint on the logical clock s and the causality (as proved in Listing 1.1 for the TCS mode transformations).

Another transformation issue arises when transforming the LTS of the CCSL constraint $a \prec_2 b \text{ delayedFor } 1 \text{ on } c_1$. This is obtained as the synchronized product

of the two constraints $a \preceq x$ (precedes) and $x \equiv b \text{ delayedFor } 1 \text{ on } c$ (coincidence), where x is an auxiliary logical clock introduced for the purpose. The LTS of both constraints are presented in Fig. 7. For the constraint $a \preceq x$, we have considered its unbounded semantics encoded by the variable d , which represents the number of instances of a that have preceded instances of x (Fig. 7.(a)). However, the ticks of x are not explicit in the final automaton presented in Fig. 6, though ‘ticks’ of both x and a update the variable d .

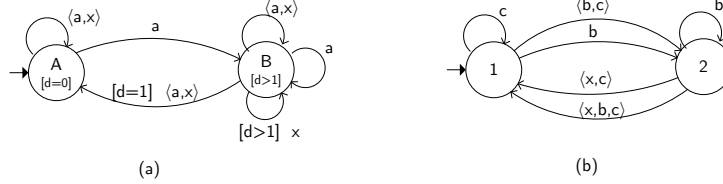


Fig. 7. LTS of CCSL constraints: a) $a \preceq x$ b) $x \equiv b \text{ delayedFor } 1 \text{ on } c$

The transformed LTS-TA for the ABS Calibrate mode is presented in Fig. 6. To make the model readable, we have not shown the update actions on d , logical clock resets on each edge (for the clocks that do not ‘tick’ in the configuration), as well as TA clock resets. To avoid the non-determinism at location XA1, we have used a priority channel P , to force the transition with guard $g3$ when both $g1$ and $g2$ also hold.

The TA mappings for the ABS, as presented above, shows that the proposed transformation addresses some of the critical issues that arise due to the expressiveness of CCSL, such as, unbounded operators, mixed constraints, and chronometric time.

5 Verification

In this section, we present a verification approach for MARTE/CCSL mode behaviors by model checking the corresponding TA, obtained using the transformation approach presented in the previous section. Verification is performed using the UPPAAL tool. A set of properties describing deadlock-freedom, liveness, causality, and chronometric time is specified and verified for the example systems. To support the verification, we use observer automata for specific kinds of properties, and extend the automata resulting from the transformation, to support synchronous (timewise) interactions with the observers. Such extensions can be easily automated together with the entire model transformation.

Deadlock-freedom. The property specified in Eqn.1, as a safety-property, describes the absence of deadlocks. A deadlock occurs when the system cannot progress further. For both TCS and ABS mode behaviors, the property is satisfied.

$$A \square \neg \text{deadlock} \quad (1)$$

Deadlock-Path identification problem for logical clocks. For CCSL specifications, one of very important and hard to achieve verification problems is the identification of the

execution paths, or sub-paths, for which a given set of clocks eventually do not ‘tick’. In CCSL, such paths are referred to as deadlock-paths for a given set of logical clocks. For instance, for the TCS Diagnostic TAn, we have verified the presence of a deadlock-path, using property in Eqn. 2, for the logical clock s (`StatusUpdate` action). The equation models a liveness property, as a “leads to” property in UPPAAL (denoted by \rightsquigarrow , implemented as $-->$ in UPPAAL). The property (2) basically states that for all paths, it is always the case that the clock will eventually tick. In the TCS example, the property fails to hold and an execution path where s never ticks eventually is shown as a counter-example/diagnostic trace. The diagnostic traces show the execution path where s never ticks. The property can be extended to multiple clocks, as in Eqn. 3, where clocks c and s correspond to `Reconfig` and `StatusUpdate` respectively. The property is satisfied, indicating that the clocks together do not lead to any deadlock paths of the Diagnostic mode executions.

$$s==0 \rightsquigarrow s==1 \quad (2)$$

$$(c==0 \ \&\& \ s==0) \rightsquigarrow (c==1 \ || \ s==1) \quad (3)$$

Chronometric durations of logical clocks and event chains. Another benefit provided by transforming mode behaviors into TA is the possibility of verifying chronometric aspects, such as, *minimum* and *maximum* inter-arrival times, (m, M) , of a logical clock with no explicit chronometric constraints otherwise. For this, we use an observer automaton as shown in Fig. 8, and the corresponding property to be verified, given by (4). To enable (time-wise) synchronous interactions between the specification automaton and the observer, we introduce in the former, between the source and the target locations, a committed location that connects to the target location through an edge annotated with the synchronization channel ‘`sig!`’, as shown in Fig. 8. The observer computes the time between two ‘ticks’ of the logical clock r . By the timing property given by (4), one is able to verify that the (min, max) inter-arrival time of r is $(0, 40)$ for the `RemoveRod` action.

$$A \square (t == 1 \ \text{imply} \ (rx > m \ \text{and} \ rx < M)) \quad (4)$$

We can generalize the observer automaton for two events, to verify end-to-end timing of event chains that consist of a stimulus-response event pair. For instance, the ABS Calibrate mode has CCSL timing constraint ‘ $1 \leq s \ \text{delayedFor} \ 10 \ \text{ms}$ ’, which specifies the end-to-end timing for s, l representing `SenseSpeed` and `Calibrate` respectively. However, the event chain may contain sub-events, which makes it necessary to verify the consistency of the constraints.

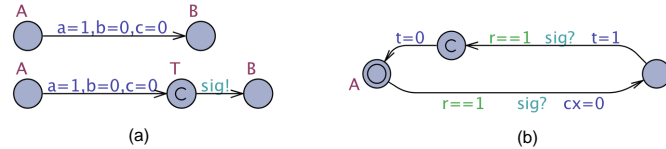


Fig. 8. a) Extending mode TAn transitions b) Observer TAn for chronometric durations.

6 Related Work

Wang et.al have recently proposed the MDM (Mode Diagram Modeling) framework for periodic control systems [16]. They have also provided a property specification language, based on the interval logic, to capture the temporal properties that can be verified against MDM models, using statistical model checking. Unlike our approach, the complete verification is undecidable, as MDM may involve complex non-linear computations. Another comparable framework is that of *Modecharts* and RTL (real time logic) [10]. RTL assertions for events are similar to CCSL constraints involving logical and chronometric clocks. Both approaches define a time structure to specify timed causality semantics of the system (CCSL is more expressive given its polychronous semantics), and provide structural organization of a system's causality and timing behavior to efficiently reason about system timing properties. In comparison, our approach provides the capability of verifying usual dense-time properties, but also combinations of logical and chronometric time properties, a feature not existing before. Several related works have proposed various transformation approaches for mapping CCSL or its subsets, into different semantic domains such as VHDL, Petri nets, and Promela. André et al. presented an automatic transformation of a CCSL specification into VHDL code [5]. The proposed transformation assembles instances of pre-built VHDL components while preserving the polychronous semantics of CCSL. The generated code can be integrated in the VHDL design and verification flow. Mallet and André have recently proposed a formal semantics to a kernel subset of MARTE, and presented an equivalent interpretation of the kernel in two different formal languages, namely Signal and Time Petri nets [13]. In their work, relevant examples have been used to show instances when Petri-nets are suitable to express CCSL constraints, as well as instances where synchronous languages are more appropriate. Ling et al. have proposed a transformation approach for logical CCSL constraints into Promela, using checkpoint-bisimulation approach, for verification with SPIN model-checker [17]. Also, some property specification patterns for expressing the properties of the model have been proposed. In comparison to above transformation based approaches, here we have proposed a model-checking based approach that addresses chronometric time constraints of CCSL effectively, by overcoming the limitations in specifying synchrony in otherwise asynchronous modeling of timed automata.

7 Conclusions and Future work

In this paper, we have proposed a transformation approach for MARTE/CCSL mode behavior specifications into timed automata, to enable model-checking of the specifications using UPPAAL. The approach is based on the synchronized product of the CCSL semantics. As a main contribution, we have been able to bridge the CCSL and timed automata based frameworks, by successfully mapping the synchronous and discrete chronometric semantics of CCSL into the asynchronous and dense time semantics of timed automata. To demonstrate the benefits of the proposed transformation approach, we have verified both logical and chronometric properties using the mode behavior specification of the example systems in this paper. Since CCSL is an expressive language, we have considered a subset of CCSL constraints for the transformation, and

plan to investigate other constraints as future work. To support the verification process, we will investigate specific classes of logical and timing properties that can be verified, and will model them as property patterns or timed automata observers. Currently, a prototype version of the tool for constructing synchronized products of CCSL constraints exists, so we intend to formalize the proposed model transformation technique and integrate UPPAAL in a MARTE/CCSL modeling framework.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *the 1st International Workshop on Formal Modeling and Analysis of Timed Systems*, May 2003.
3. C. André, F. Mallet, and R. de Simone. Modeling time(s). In *Models'07*, volume 4735 of *LNCS*, pages 559–573. Springer, 2007.
4. Charles André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Rapport de recherche RR-6925, INRIA, 2009.
5. Charles André, Frédéric Mallet, and Julien DeAntoni. VHDL observers for clock constraint checking. In *Industrial Embedded Systems (SIES), 2010 Int. Symp. on*, pages 98–107, July.
6. André Arnold. *Finite transition systems - semantics of communicating systems*. Int. Series in Computer Science. Prentice Hall, 1994.
7. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
8. B. Bouyssounouse and J. Sifakis. *Embedded Systems Design: The ARTIST Roadmap for Research and Development (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
9. Julien Deantoni and Frédéric Mallet. TimeSquare: Treat your Models with Logical Time. In *TOOLS - 50th Int. Conf. on Objects, Models, Components, Patterns*, volume 7304 of *LNCS*, pages 34–41. Springer, May 2012. Available at <http://timesquare.inria.fr>.
10. F. Jahanian and A.K. Mok. Modechart: a specification language for real-time systems. *Software Engineering, IEEE Transactions on*, 20(12):933–947, Dec.
11. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
12. Frédéric Mallet. Automatic Generation of Observers from MARTE/CCSL. In *Int. Symp. on Rapid System Prototyping - RSP 2012*, Tampere, Finland, 2012. IEEE.
13. Frédéric Mallet and Charles André. On the semantics of UML/Marte Clock Constraints. In *Int. Symp. on Object/component/service-oriented Real-time distributed Computing (ISORC'09)*, pages 301–312, Tokyo, Japon, 2009. IEEE.
14. Object Management Group (OMG). UML 2.0 Superstructure Specification, The OMG Final Adopted Specification, 2003.
15. OMG. *UML Profile for MARTE, v1.0*. Object Management Group, November 2009. formal/2009-11-02.
16. Zheng Wang, Geguang Pu, Jianwen Li, Jifeng He, Shengchao Qin, Kim G. Larsen, Jan Madsen, and Bin Gu. Mdm: A mode diagram modeling framework. In *Proc. First International Workshop on Formal Techniques for Safety-Critical Systems*, EPTCS, pages 135–149, 2012.
17. Ling Yin, Frédéric Mallet, and Jing Liu. Verification of MARTE/CCSL time requirements in Promela/SPIN. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE Int. Conf. on*, pages 65–74, April.