Logical time and temporal logics: comparing UML MARTE/CCSL and PSL

Régis Gascon, Frédéric Mallet, Julien Deantoni

Logical time and temporal logics: comparing UML MARTE/CCSL and PSL

Régis Gascon, Frédéric Mallet, Julien DeAntoni
AOSTE Project, I3S/INRIA
Université Nice Sophia-Antipolis & INRIA Sophia-Antipolis Méditerranée
e-mail: Frederic.Mallet@unice.fr

***Abstract*—The UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) has been recently adopted. The Clock Constraint Specification Language (CCSL) allows the specification of causal, chronological and timed properties of MARTE models. Due to its purposely broad scope of use, CCSL has an expressiveness that can prevent formal verification. However, when addressing hardware electronic systems, formal verification is an important step of the development. The IEEE Property Specification Language (PSL) provides a formal notation for expressing temporal logic properties that can be automatically verified on electronic system models. In this paper, we determine the part of MARTE/CCSL amenable to support the classical analysis methods from the Electronic Design Automation (EDA) community by comparing CCSL and PSL expressiveness. We show that neither of these languages is subsumed by the other one. We identify and restrict the CCSL constructs that cannot be expressed in temporal logics so that CCSL become tractable in temporal logics. Conversely, we also identify the class of PSL formulas that can be encoded in CCSL. We define translations between these fragments of CCSL and PSL using automata as an intermediate representation.**

***Keywords*-High-level design, Linear temporal logic, Language equivalence, Automaton based approach.**

I. Introduction

The UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE [8]) provides a means to specify several aspects of embedded systems, ranging from large software systems on top of an operating system to specific hardware designs. UML/MARTE provides a support to capture structural and behavioral, functional and non-functional aspects. The Clock Constraint Specification Language (CCSL [1]), initially specified in an annex of MARTE, offers a general set of notations to specify causal, chronological and timed properties on these models and has been used in various subdomains [6], [2]. CCSL is formally defined and CCSL specifications can be executed at the model level. CCSL is intended to be used at various modeling levels following a refinement strategy. It should allow both coarse, possibly non-deterministic, infinite, unbounded specifications at the system level but also more precise specifications from which code generation, schedulability and formal analysis are possible.

In the domain of hardware electronic systems, which is one of the subdomains targeted by MARTE, formal verification is an important step of the development. To allow simulation and formal verification of such systems, the IEEE Property Specification Language (PSL [10]) provides a formal notation for the specification of electronic system behavior, compatible with multiple electronic system design languages (VHDL, Verilog, SystemC, SystemVerilog).

In a Model-Driven approach where code (*e.g.,*SystemC or VHDL) is generated from models (UML/MARTE), two questions arise. Is MARTE expressive enough to capture an abstract view of hardware systems? Is CCSL expressive enough to express properties usually modeled in PSL? Some efforts have been made to answer the first question [9], [12]. We are addressing here the second question and we focus on properties expressed with CCSL on top of MARTE models.

The main contribution of this paper is the comparison of PSL and CCSL expressiveness. The first result is that neither of these languages subsume the other one. Consequently, we identify the CCSL constructs that cannot be expressed in temporal logics and propose restrictions to these operators so that they become tractable in temporal logics. Conversely, we also identify the class of PSL formulas that can be encoded in CCSL. Using this information, we show that translations between large fragments of CCSL and PSL can be defined. Because direct modular translation is more tedious, we use an automaton-based approach. Though translation from PSL to automata is a well studied topic (see e.g., [3]), similar transformation for CCSL specifications is new and interesting result. This intermediate translation of CCSL specifications to automata could be alternatively used directly by the subdomain tools (and possibly completed by other PSL properties if needed). However, the main purpose of the paper is the comparison of PSL and CCSL and we do not claim that the rigorous translation chains we define is adequate to perform fast analyses.

The rest of this paper is organized as follows. In Sect. II we introduce CCSL and PSL and determine which kind of properties cannot be expressed in each language. We define in Sect. III the class of Boolean automata which is used in Sect. IV to define translations between fragments of CCSL and PSL. Sect. V contains concluding remarks and future work. Because of space limitation, some proofs and technical details are omitted but can be found in [4].

II. Definitions of the languages

We firt define the languages that we consider and give preliminary comparisons related to their expressive power.

A. Clock Constraint Specification Language

CCSL is the companion language of the UML MARTE profile for the design of embedded systems. It combines constructs from the general net theory and from synchronous languages. CCSL offers a set of causal and timed patterns classically used in embedded systems. More formally, CCSL is based on the notion of *clocks* which is a general name to denote a totally ordered sequence of event occurrences, called the *instants* of the clock. Instants do not carry values. CCSL defines a set of *clock relations*:

$$r ::= c_1 \boxed{\subset} c_2 \mid c_1 \boxed{\#} c_2 \mid c_1 \boxed{\prec} c_2 \mid c_1 \boxed{\preccurlyeq} c_2$$

where c_1, c_2 represent clocks of the system. Informally, $c_1 \boxed{\subset} c_2$ means that c_1 is a subclock of c_2, $c_1 \boxed{\#} c_2$ that the instants of the two clocks never coincide and $c_1 \boxed{\preccurlyeq} c_2$ (resp.$c_1 \boxed{\prec} c_2$) that the n^{th} occurrence of c_1 precedes (resp. strictly precedes) the n^{th} occurrence of c_2 for every $n \in \mathbb{N}^*$ ($\mathbb{N}^*$ denotes the set of strictly positive integers).

CCSL is a high-level multiclock language and the original semantics does not require totally ordered models. However, at lower level or for simulation purposes, one needs to represent the execution as a totally ordered sequence. In this context, the alternative operational semantics introduced in [1] identifies clocks with Boolean variables evolving along time. In the remaining, we will consider that a clock c belongs to a set of propositions VAR and CCSL models are finite or infinite sequences of elements in 2^{VAR}. The set of instants of the clock c corresponds to the set of positions where the variable c holds.

Let σ be a CCSL model. For such a sequence, we denote by $|\sigma|$ the length of σ and assume that $|\sigma| = \omega$ when σ is an infinite word. We use the notations $\sigma(i)$ for the i^{th} element of σ and σ^i for the suffix of σ starting at the i^{th} position. To evaluate the satisfaction of precedence relations, we define the function χ_σ that counts the number of times a clock c occurs in the i^{th} first positions of σ, i.e.

$$\chi_\sigma(c, i) = |\{j \in \mathbb{N} \text{ s.t. } j \leq i \text{ and } c \in \sigma(j)\}|.$$

The satisfaction of CCSL relations is defined by:

- $\sigma \models_{ccsl} c_1 \boxed{\subset} c_2$ iff for every $0 \leq i < |\sigma|$, if $c_1 \in \sigma(i)$ then $c_2 \in \sigma(i)$.
  We also note $\sigma \models_{ccsl} c_1 \boxed{=} c_2$ iff $\sigma \models_{ccsl} c_1 \boxed{\subset} c_2$ and $\sigma \models_{ccsl} c_2 \boxed{\subset} c_1$.
- $\sigma \models_{ccsl} c_1 \boxed{\#} c_2$ iff for every $0 \leq i < |\sigma|$ we have $c_1 \notin \sigma(i)$ or $c_2 \notin \sigma(i)$.
- $\sigma \models_{ccsl} c_1 \boxed{\prec} c_2$ iff for every $0 \leq i < |\sigma|$ we have $c_2 \in \sigma(i)$ implies $\chi_\sigma(c_1, i-1) > \chi_\sigma(c_2, i-1)$.
- $\sigma \models_{ccsl} c_1 \boxed{\preccurlyeq} c_2$ iff for every $0 \leq i < |\sigma|$ we have $\chi_\sigma(c_1, i) \geq \chi_\sigma(c_2, i)$.

CCSL can also express more complicated constraints between clocks using *clock definitions*. CCSL clock definitions allow the definition of a clock as the combination of other clocks given as arguments. A clock definition is of the form $c \boxed{\triangleq} e$ where $c \in \text{VAR}$ and e is a *clock expression* defined by the following grammar:

$$e := c \mid e + e \mid e * e \mid e \searrow e \mid e \searrow\!\!\searrow e \mid e \lightning e \mid e \blacktriangledown bw \mid e \$_e\, n \mid e \vee e \mid e \wedge e$$

where $c \in \text{VAR}$, $n \in \mathbb{N}^*$ and $bw : \mathbb{N}^* \to \mathbb{B}$ is an ultimately periodic binary word (of the form $u \cdot v^\omega$). The expressions $e_1 + e_2$ and $e_1 * e_2$ represent respectively the union and intersection of e_1 and e_2. The strict and non-strict sample expressions are denoted by $e_1 \searrow e_2$ and $e_1 \searrow\!\!\searrow e_2$. The delay operation $e_1 \$_{e_2}\, n$ is a variation of sampling that samples e_1 on the n^{th} occurrence of e_2. The expression $e_1 \lightning e_2$ is the preemption (e_1 up to e_2), $e \blacktriangledown bw$ represents the filtering operation. Finally, $e_1 \vee e_2$ (resp. $e_1 \wedge e_2$) represents the fastest (resp. slowest) of the clocks that are slower (resp. faster) than both e_1 and e_2. This corresponds to greatest lower bound and lowest upper bound.

Given a clock expression e and a CCSL model σ we note $\sigma, i \models_{ccsl} e$ iff the expression e holds at position i of σ.

- $\sigma, i \models_{ccsl} c$ iff $c \in \sigma(i)$.
- $\sigma, i \models_{ccsl} e_1 + e_2$ iff $\sigma, i \models_{ccsl} e_1$ or $\sigma, i \models_{ccsl} e_2$.
- $\sigma, i \models_{ccsl} e_1 * e_2$ iff $\sigma, i \models_{ccsl} e_1$ and $\sigma, i \models_{ccsl} e_2$.
- $\sigma, i \models_{ccsl} e_1 \searrow e_2$ iff
  - $\sigma, i \models_{ccsl} e_2$,
  - there is $0 \leq j < i$ such that $\sigma, j \models_{ccsl} e_1$ and for every $j \leq k < i$ we have $\sigma, k \not\models_{ccsl} e_2$.
- $\sigma, i \models_{ccsl} e_1 \searrow\!\!\searrow e_2$ iff
  - $\sigma, i \models_{ccsl} e_2$,
  - there is $0 \leq j \leq i$ such that $\sigma, j \models_{ccsl} e_1$ and for every $j \leq k < i$ we have $\sigma, k \not\models_{ccsl} e_2$.
- $\sigma, i \models_{ccsl} e_1 \$_{e_2}\, n$ there exists $0 \leq j \leq i$ such that
  - $\sigma, j \models_{ccsl} e_1$ and
  - there are exactly n distinct positions $i_1, \ldots, i_n$ ($i_n = i$) such that for every $k \in \{1, \ldots, n\}$ we have $j < i_k \leq i$ and $\sigma, i_k \models_{ccsl} e_2$.
- $\sigma, i \models_{ccsl} e_1 \lightning e_2$ iff
  - $\sigma, i \models_{ccsl} e_1$,
  - for every $0 \leq j \leq i$ we have $\sigma, j \not\models_{ccsl} e_2$.
- $\sigma, i \models_{ccsl} e \blacktriangledown bw$ iff
  - $\sigma, i \models_{ccsl} e$
  - $bw(\chi_\sigma(e, i)) = 1$.
- $\sigma, i \models_{ccsl} e_1 \wedge e_2$ iff either
  - $\chi_\sigma(e_1, i) > \chi_\sigma(e_2, i)$ and $\sigma, i \models_{ccsl} e_1$,
  - or $\chi_\sigma(e_1, i) < \chi_\sigma(e_2, i)$ and $\sigma, i \models_{ccsl} e_2$,
  - or $\chi_\sigma(e_1, i) = \chi_\sigma(e_2, i)$ and $\sigma, i \models_{ccsl} e_1$ and $\sigma, i \models_{ccsl} e_2$.
- $\sigma, i \models_{ccsl} e_1 \vee e_2$ iff either
  - $\chi_\sigma(e_1, i) > \chi_\sigma(e_2, i)$ and $\sigma, i \models_{ccsl} e_2$,
  - or $\chi_\sigma(e_1, i) < \chi_\sigma(e_2, i)$ and $\sigma, i \models_{ccsl} e_1$,
  - or $\chi_\sigma(e_1, i) = \chi_\sigma(e_2, i)$ and we have $\sigma, i \models_{ccsl} e_1$ or $\sigma, i \models_{ccsl} e_2$.

The function χ_σ above is extended to expressions in a natural way:

$$\chi_\sigma(e,i) = |\{j \in \mathbb{N} \text{ s.t. } j \leq i \text{ and } \sigma, j \models_{ccsl} e\}|.$$

A CCSL specification is a list of definitions and relations seen as a conjunction of constraints. We can represent it by a triple $\langle C, Def, Rel\rangle$ such that $C \subseteq \text{VAR}$ is a set of clocks, Def is a set of definitions, Rel is a set of relations. A model σ over 2^C satisfies the specification iff for every definition $c \triangleq e$ in Def we have $c \in \sigma(i)$ iff $\sigma, i \models_{ccsl} e$, and every relation in Rel is satisfied by σ.

From the basis CCSL language, one can define other expressions and relations. For instance, the following expressions and relations are used later:

- $c_1 - c_2$ is the difference of clocks c_1 and c_2. The definition $c \triangleq c_1 - c_2$ can be encoded with the definition $c_1 \triangleq c + c_2$ and the relation $c \# c_2$.
- $c \$_c n$ is a particular case of delay expression that we shortly note $c \$ n$. It represents the usual synchronous delay operation. The resulting expression starts at the n^{th} occurrence of c and then coincides with c.
- Right weak alternation $c_1 \tilde{=} c_2$ is defined by the relations $c_1 \preccurlyeq c_2$ and $c_2 \prec c_1'$ where $c_1' \triangleq c_1 \$ 1$. Similarly, left weak alternation $c_1 \tilde{=} c_2$ is defined by $c_1 \preccurlyeq c_2$ and $c_2 \prec c_1'$.

B. Property Specification Language

The IEEE standard PSL [10] has been designed to provide an interface to hardware formal verification. Its temporal layer is a textual language to build temporal logic expressions. PSL assertions can then be validated by model-checking or equivalence checking techniques. The underlying linear-time logic in PSL extends LTL with regular expressions and sugaring constructs. However PSL remains as expressive as ω-regular languages. As it would be tedious to consider the sugaring operators of PSL in formal reasoning, we use the minimal core language defined in [3].

Let VAR be a set of propositions (Boolean variables) that aims at representing signals of the system. PSL atomic formulas are called *Sequential Extended Regular Expressions* (SERE). SEREs are basically regular expressions built over the Boolean algebra:

$$b ::= x \mid \overline{x} \mid b \wedge b \mid b \vee b$$

where $x \in \text{VAR}$ is a Boolean variable. We also consider the standard implication and equivalence operators $\Rightarrow$ and $\Leftrightarrow$ that can be defined from the grammar above. The set of SEREs is defined by:

$$r ::= b \mid r \cdot r \mid r \cup r \mid r^*$$

where b is a Boolean formula. The operators have their usual meaning: $r_1 \cdot r_2$ is the concatenation, $r_1 \cup r_2$ the union and r^* is the Kleene star operator. From these regular expressions, PSL linear properties are defined by:

$$\phi ::= r \mid \phi \wedge \phi \mid \neg\phi \mid \mathsf{X}\phi \mid \phi \mathsf{U} \phi \mid r \rightarrowtail \phi.$$

where r is a SERE. The operators X (next) and U (until) are the classical temporal logic operators. We also use the classical abbreviations $\mathsf{F}\phi \equiv \top \mathsf{U} \phi$ (eventually) and $\mathsf{G}\phi \equiv \neg\mathsf{F}\neg\phi$ (always). The formula $r \rightarrowtail \phi$ is a "suffix conjunction" operator meaning that there must exist a finite prefix satisfying r and that ϕ must be satisfied at the position corresponding to the end of this prefix (with a one-letter overlap between the prefix and the suffix).

The semantics of PSL is defined in such a way that properties can be interpreted over infinite words as well as finite or truncated words. This is important for applications like simulation or bounded model-checking. Similarly to CCSL, the models of PSL are finite or infinite sequences over elements of 2^{VAR}.

For every $X \in 2^{\text{VAR}}$ and $p \in \text{VAR}$, we note $X \models_b p$ iff $p \in X$ and $X \models_b \overline{p}$ iff $p \notin X$. The remaining of the Boolean satisfaction relation $\models_b$ are standard. SEREs refer to a finite (possibly empty) prefix of the model. So σ is supposed to be finite in the SERE satisfaction relation (which is not the case in the PSL satisfaction relation). The SERE satisfaction is defined by induction as follows:

- $\sigma \models_{re} b$ iff $|\sigma| = 1$ and $\sigma(0) \models_b b$,
- $\sigma \models_{re} r_1 \cdot r_2$ iff there are σ_1, σ_2 such that $\sigma = \sigma_1\sigma_2$ and $\sigma_1 \models_{re} r_1$ and $\sigma_2 \models_{re} r_2$.
- $\sigma \models_{re} r_1 \cup r_2$ iff $\sigma \models_{re} r_1$ and $\sigma \models_{re} r_2$.
- $\sigma \models_{re} r^*$ iff either $\sigma = \varepsilon$ (empty sequence) or there exist $\sigma_1 \neq \varepsilon$ and σ_2 such that $\sigma = \sigma_1\sigma_2$, $\sigma_1 \models_{re} r$ and $\sigma_2 \models_{re} r^*$.

Finally, the satisfaction of PSL properties is defined by:

- $\sigma \models_{psl} \neg\phi$ iff $\sigma \not\models_{psl} \phi$,
- $\sigma \models_{psl} \phi_1 \wedge \phi_2$ iff $\sigma \models_{psl} \phi_1$ and $\sigma \models_{psl} \phi_2$,
- $\sigma \models_{psl} \mathsf{X}\phi$ iff $|\sigma| > 1$ and $\sigma^1 \models_{psl} \phi$,
- $\sigma \models_{psl} \phi_1 \mathsf{U} \phi_2$ iff there is $0 \leq i < |\sigma|$ such that
  - $\sigma^i \models_{psl} \phi_2$ and
  - for every $0 \leq j < i$ we have $\sigma^j \models_{psl} \phi_1$,
- $\sigma \models_{psl} r \rightarrowtail \phi$ iff there is a finite prefix $\sigma_1\alpha$ of σ ($\alpha \in 2^{\text{VAR}}$ is a single letter) such that
  - $\sigma = \sigma_1\alpha\sigma_2$,
  - $\sigma_1\alpha \models_{re} r$ and
  - $\alpha\sigma_2 \models_{psl} \phi$,
- $\sigma \models_{psl} r$ iff for every finite prefix σ_1 of σ there is a finite word σ_2 such that $\sigma_1\sigma_2 \models_{re} r \rightarrowtail \top$.

C. Comparing PSL and CCSL

The CCSL semantics we consider in this paper is restricted. In the general definition, models of CCSL specifications do not need to be totally ordered. However, under this restriction CCSL and PSL share common models. So we can compare the classes of properties they can express.

Let S be a CCSL specification over a set of variables $V_S \subseteq \text{VAR}$ and ϕ a PSL formula over a set of variables $V_\phi \subseteq \text{VAR}$. We will say that S is encoded by (or simulated by) ϕ such that $V_S \subseteq V_\phi$ iff every model of ϕ is also a model of S and every model of S can be extended on V_ϕ to a model of ϕ. The converse simulation relation is a bit different. CCSL models have the properties that one can add an unbounded amount of empty states between two relevant states and leave the satisfaction unchanged. This can easily be proved by induction on the structure of a CCSL specification.

Lemma 1. Let S be a CCSL specification. For all model σ satisfying S and every $0 \leq i \leq |\sigma|$ the model σ' defined by

$$\begin{aligned}&\sigma'(j) = \sigma(j) \text{ for every } j < i\\ &\sigma'(i) = \emptyset\\ &\sigma'(j) = \sigma(j-1) \text{ for every } i < j \leq |\sigma| + 1\end{aligned}$$

also satisfies S.

This property is a consequence of the multiclock aspect of CCSL. It is not possible to completely link the execution of a CCSL specification to a global clock. However, the states where no clocks occur are *irrelevant* in the CCSL point of view as they do not make the system evolve. So it is not really a problem to discard them. We will say that ϕ is simulated by S such that $V_\phi \subseteq V_S$ iff every model of S *with no irrelevant states* is also a model of ϕ and every model of ϕ can be extended to a model of S.

Some CCSL relations or expressions implicitly introduce unbounded counters. For instance, one has to store the number of occurrences of the clocks c_1 and c_2 (or the difference between them) to encode the precedence relation $c_1 \preccurlyeq c_2$. The corresponding language is made of all the words such that every finite prefix contains more occurrences of c_1 than c_2. Such a language is neither regular nor ω-regular and cannot be encoded in PSL. The same remark holds for the expressions $c_1 \wedge c_2$ and $c_1 \vee c_2$. On the other hand, CCSL relations and expressions only state safety constraints. As a specification is a conjunction of such constraints, the result is always a safety property. CCSL cannot express liveness like the reachability property $\mathsf{F}p$. For finite executions, there is also no way to express that the model must have a next position which can be stated by $\mathsf{X}\top$ in PSL. To summarize, the preliminary comparison of expressiveness of CCSL and PSL gives the following results.

Theorem 1. **(I)** There are PSL formulas that cannot be encoded in CCSL. **(II)** There are CCSL specifications that cannot be encoded in PSL.

It is now clear that PSL and CCSL are not comparable in their whole definition. We show in the remainder that restricting the properties of each language according to the observations above is enough to obtain large fragments that can express the same set of properties. We define a method to encode a property in each of these fragments into the other, which is based on intermediate translation into automata.

III. Boolean automata

We introduce in this section the class of automata we use to define relations between PSL and CCSL fragments.

A. Definition

We consider automata that handle propositional variables in VAR. The transitions are labeled by Boolean formulas interpreted like guards. Formally, a *Boolean automaton* is a structure $\mathcal{A} = \langle Q, q_0, F, A, V, \delta\rangle$ such that Q is a set of states and $q_0 \in Q$ an initial state, $F \subseteq Q$ and $A \subseteq Q$ are respectively the set of final and accepting states, $V \subseteq \text{VAR}$ is a set of propositions, $\delta : Q \times Bool(V) \times Q$ is a transition relation where $Bool(V)$ is the set of Boolean formulas over V. We use the definitions of Sect. II-B for Boolean formulas. A Boolean automaton is deterministic iff for every state in Q there do not exist two outgoing transitions labeled with ϕ and ϕ' such that $\phi \wedge \phi'$ is satisfiable.

A configuration of $\mathcal{A}$ is a pair $\langle q, X\rangle$ composed of a state in Q and a subset of V. We note $\langle q, X\rangle \xrightarrow{\phi} \langle q', X'\rangle$ iff there is a transition $q \xrightarrow{\phi} q'$ such that $X \models_b \phi$. A run of $\mathcal{A}$ is a sequence $\sigma : \mathbb{N} \to (Q \times 2^V)$ such that $\sigma(0)$ is of the form $\langle q_0, X_0\rangle$ (one starts in the initial state) and for every $i \in \mathbb{N}$, there exists ϕ_i such that $\sigma(i) \xrightarrow{\phi_i} \sigma(i+1)$. A finite run is accepting iff it ends in a final state. An infinite run is accepting iff it visits infinitely often an accepting state (Büchi condition). The language accepted by $\mathcal{A}$ is the set of words on the alphabet 2^V corresponding to accepting runs.

Let $\mathcal{A}_1 = \langle Q_1, (q_0)_1, F_1, A_1, V_1, \delta_1\rangle$ and $\mathcal{A}_2 = \langle Q_2, (q_0)_2, F_2, A_2, V_2, \delta_2\rangle$ be two Boolean automata. The product automaton $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2$ is the structure $\langle Q, q_0, F, A, V, \delta\rangle$ such that:

- $Q = Q_1 \times Q_2 \times \{0, 1\}$ where the component in $\{0, 1\}$ is only needed for the Büchi acceptance condition,
- $V = V_1 \cup V_2$,
- $q_0 = \langle (q_0)_1, (q_0)_2, 0\rangle$,
- $F = F_1 \times F_2 \times \{0, 1\}$ and $A = Q_1 \times A_2 \times \{1\}$,
- For every $\langle q_1, q_2, i\rangle$ and $\langle q_1', q_2', i'\rangle$ in Q we have $\langle q_1, q_2, i\rangle \xrightarrow{\phi} \langle q_1', q_2', i'\rangle$ iff
  - there exist $q_1 \xrightarrow{\phi_1} q_1'$ and $q_2 \xrightarrow{\phi_2} q_2'$ such that ϕ is equivalent to $\phi_1 \wedge \phi_2$,
  - if $i = 0$ then $i' = 1$ iff $q_1 \in A_1$,
  - if $i = 1$ then $i' = 0$ iff $q_2 \in A_2$.

Note that the last component of each state is not needed when every state is accepting ($A_1 = Q_1$ and $A_2 = Q_2$).

B. CCSL and Boolean automata

Since CCSL expresses only safety, the acceptance condition of automata cannot be encoded. However, if every run is accepting we can encode a deterministic Boolean automaton into a CCSL specification.

Lemma 2. Every deterministic Boolean automaton such that every execution is accepting can be simulated by a CCSL specification.

Proof: Consider a deterministic Boolean automaton $\mathcal{A} = \langle Q, q_0, V, \delta \rangle$. We forget accepting and final states since every valid execution is accepted. We define the set of clocks $C = V \uplus Q$. To encode $\mathcal{A}$, we need the following CCSL definitions. We define a global clock and a clock corresponding to the set of states Q as follows:

$$(\mathbf{1}) \quad \mathit{Glob} \triangleq \sum_{c \in C} c \quad \text{and} \quad Q \triangleq \sum_{q \in Q} q$$

where $\sum_{c \in X} c$ is the CCSL union of all the clocks in X. Similarly, we note $\prod_{c \in X} c$ the CCSL intersection of all the clocks in X. For ease of presentation, we note $q \xrightarrow{X} q'$ iff there is a transition $q \xrightarrow{\phi} q'$ in $\mathcal{A}$ such that $X \models_b \phi$. For every state $q \in Q \setminus \{q_0\}$, we define the clock Iq corresponding to the incoming transitions of q:

$$(\mathbf{2}) \quad Iq \triangleq \sum_{q' \xrightarrow{X} q} \big(q' * (\prod_{p \in X} p) - (\sum_{p \notin X} p)\big).$$

Now we build the set of CCSL relations. First we express that at every position in the run, exactly one state of the automaton holds. This corresponds to the relations

$$(\mathbf{3}) \quad Q \mid = \mid \mathit{Glob} \text{ and } q \mid \# \mid q' \text{ for all } q, q' \in Q \ (q \neq q').$$

We also impose that the global clock always coincides with a valid transition in order to avoid unexpected behaviors:

$$(\mathbf{4}) \quad \mathit{Glob} \mid = \mid \mathit{Trans} \quad \text{where} \quad \mathit{Trans} \triangleq \sum_{q \in Q} I_Q.$$

The transition relation is such that every state alternates with its incoming transitions. So, for every $q \in Q$

$$(\mathbf{5}) \quad q_0 \mid \sim \mid Iq_0 \quad \text{and} \quad Iq \mid \sim \mid q.$$

The relation is symmetric for q_0 since the execution starts in this state. The alternation is not strict on the side of the incoming transition since it is allowed to return immediatly to the same state (self loop).

We now have to show that a model σ satisfies the CCSL specification obtained iff there is a run ρ of $\mathcal{A}$ such that for every $0 \leq i \leq |\sigma|$, for every $c \in V$ we have $c \in \sigma(i)$ iff $\rho(i) = \langle q_i, X_i \rangle$ and $c \in X_i$. We consider only runs where at least one variable holds at each position. They are the only relevant runs according to the simulation relation. By construction, in such runs the clock Glob always holds. So we can identified the steps with the occurrences of Glob.

We need intermediate properties to show that each execution corresponds to a run of $\mathcal{A}$. Let C_I be the set of clocks of the form Iq for every q. Since the clocks in C_I are defined wrt. a deterministic transition relation (cf. **(2)**) and at each step exactly one element of Q holds (by **(3)**), we have at most one element of C_I occurring at each step. Combined with **(4)**, this implies that if the current state is not a deadlock then there is exactly one element of C_I and one of Q belonging to $\sigma(i)$ for every $0 \leq i < |\sigma|$. In presence of deadlock, the CCSL execution also deadlocks. By hypothesis, the corresponding finite run is accepted and there is no infinite run possible from this prefix.

The property above allows us to prove that for every $0 \leq i < |\sigma|$ if $Iq \in \sigma(i)$ then $q \in \sigma(i+1)$. At the first position, the only state that can occur is q_0. Indeed, **(5)** prevents the other states since no clock of C_I has occurred yet. Suppose that $Iq \in \sigma(0)$ (Iq is unique, see above). At the next position the situation is the following (as consequences of **(5)**):

- q_0 has occurred and cannot occur until Iq_0 has occurred. Note that Iq_0 can occur at the first position since the alternance relation is weak.
- every other state q' can occurs only if Iq' has occurred at the first position.

So the only state that can occur at the second step is q. The induction step is similar. Just replace q_0 by the state that holds at the current position. Indeed, every clock that does not occur is still waiting for the next occurrence of the clock corresponding to its incoming transition.

It is now easy to show that σ corresponds to a run of $\mathcal{A}$. We have already proved that for every execution q_0 always holds at the first position (cf. **(5)**). Moreover, at each position i exactly one $Iq'_i \in C_I$ and one $q_i \in Q$ hold. By construction, it is obvious that $q_i \xrightarrow{X_i} q'_i$ where $X_i = \sigma(i) \cap V$ is a valid transition in $\mathcal{A}$ (recall that Iq'_i encodes q'_i incoming transitions). Moreover, we have shown that $Iq'_i \in \sigma(i)$ implies that $q'_i \in \sigma(i+1)$. So $q_{i+1} = q'_i$. By in induction, this proves that σ encodes a run of $\mathcal{A}$.

Conversely, let ρ be a run of $\mathcal{A}$. We suppose that the property holds until position i, $\rho(i) = \langle q_i, X_i \rangle$ and $q_i \in \sigma(i)$. The demonstration is symmetrical. There is a unique transition $q_i \xrightarrow{\phi} q_{i+1}$ such that $X_i \models \phi$ because the transition relation is deterministic and complete. Let set $\sigma(i)$ such that for every $c \in \mathrm{VAR}$ we have $c \in \sigma(i)$ iff $c \in X_i$. By construction, we must have $Iq_{i+1} \in \sigma(i)$ and so $q_{i+1} \in \sigma(i+1)$. Thus, one can build by induction σ verifying the property. ■

The converse translation is not possible. Index dependent relations or operators like precedence cannot be encoded by using finite state systems (see Sect. II-C).

C. PSL and Boolean automata

It is well known that one can build a finite automaton or a Büchi automaton that accepts respectively the finite and infinite models of a given PSL formula. Given a formula ϕ, the construction defined in [3] can easily be adapted to build a Boolean automaton accepting the set of models of ϕ.

Lemma 3. From any PSL property ϕ one can build a Boolean automata $\mathcal{A}_\phi$ such that the language accepted by $\mathcal{A}$ is exactly the set of models of ϕ.

The converse translation is easy since the definition of LTL is included in PSL. A construction similar to [11] allows the encoding of a Boolean automaton into an LTL formula.

Lemma 4. From any Boolean automaton $\mathcal{A}$, one can build a PSL formula $\phi_{\mathcal{A}}$ such that the set of models of $\phi_{\mathcal{A}}$ encodes the set of runs of $\mathcal{A}$.

The statement above means that each model of $\phi_{\mathcal{A}}$ is accepted by $\mathcal{A}$ and each accepting run of $\mathcal{A}$ can be extended to a model of $\phi_{\mathcal{A}}$. Indeed, the construction of $\phi_{\mathcal{A}}$ uses additional variables to encode the states of $\mathcal{A}$. The extension of a run of $\mathcal{A}$ into a model of $\phi_{\mathcal{A}}$ is straightforward considering this information.

IV. Translations for CCSL and PSL fragments

We identify in this section large fragments of CCSL and PSL that can be simulated in each other. We define translations between these fragments using intermediate Boolean automata encodings.

A. From PSL to CCSL

Lemma 2 states that deterministic Boolean automata can be encoded in CCSL when every run is accepting. Thus we restrict ourselves to the class of PSL formulas that can be translated into this subclass of Boolean automata. We consider the safety fragment of PSL defined similarly to [5] by restricting the use of negations. A PSL formula belongs to the set of *safety PSL* formulas iff (S1) subformulas of the form $\phi_1 \mathsf{U} \phi_2$ and $r \rightarrowtail \phi$ never occur under an even number of negations, and (S2) SEREs never occur under an odd number of negations. For the finite case, we also have to restrict the definition of the next operator to its weak variant (the formula is satisfied also if the model has no next position).

Lemma 5. For every property ϕ in safety PSL, one can build a deterministic Boolean automaton such that every run is accepting which accepts exactly the set of models of ϕ.

The proof is a variant of the construction in [3] (see the details in [4]). We just have to ensure that every execution is accepting. By Lemmas 5 and 2 we can encode every safety PSL formula into CCSL specifications.

Theorem 2. Every safety PSL formula can be encoded by a CCSL specification.

For example, let us consider the PSL formula $\mathsf{G}(p_0 \Rightarrow \neg(\overline{p}_1 \mathsf{U}(\overline{p}_1 \wedge p_2)))$ meaning that "there is always p_1 in an interval starting with p_0 and ending with p_2". Fig. 1 represents a Boolean automaton recognizing the models of this formula. Every state of this automaton is accepting and final. This automaton corresponds to the CCSL specification $\langle C, Def, Rel \rangle$ such that,

$Def = \{\ Q \triangleq q_0 + q_1, \quad Glob \triangleq Q + p_0 + p_1 + p_2,$
$Iq_0 \triangleq \big((q_0 - p_0) + (q_0 * p_0 * p_1) + (q_1 * p_1)\big),$

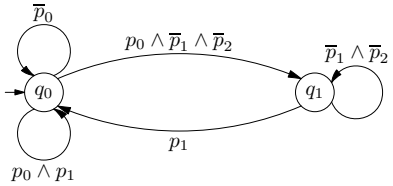

Figure 1. Boolean automaton

$Iq_1 \triangleq ((q_0 * p_0) - (p_1 + p_2)) + (q_1 - (p_1 + p_2))$
$Trans \triangleq Iq_0 + Iq_1\ \}$

and $Rel = \{\ Glob = Q, \quad q_0 \# q_1, \quad Glob = Trans,$
$q_0 \mathrel{=\!\!\sim} Iq_0, \quad Iq_1 \mathrel{\sim\!\!=} q_1\ \}.$

The set C contains all the clocks used in these different expressions and relations.

B. From CCSL to PSL

To obtain a fragment of CCSL that can be encoded in PSL, we restrict the precedence relations and the operators $c_1 \wedge c_2$ and $c_1 \vee c_2$. We define precedence relations such that the advance of the fastest clock is bounded. We denote these relations $\prec_n$ and $\preccurlyeq_n$ where $n \in \mathbb{N}$. A model σ satisfies $c_1 \prec_n c_2$ iff for all $i \in \mathbb{N}$ we have $\chi_\sigma(c_2, i) < \chi_\sigma(c_1, i) \leq \chi(c_2, i) + n$. The relation $\preccurlyeq_n$ is defined similarly with non-strict inequalities. We define similar variants $c_1 \wedge_n c_2$ and $c_1 \vee_n c_2$ restricting the difference between the clocks c_1 and c_2 to be bounded by n.

We call *bounded CCSL* the language obtained by replacing in CCSL the precedence relations, greatest lower bound and lowest upper bound operators by their bounded variants. This language is subsumed by CCSL. Indeed, the operators can be defined in CCSL:

- $c_1 \prec_n c_2$ is equivalent to $c_1 \prec c_2$ and $c_2 \prec c_1'$ where $c_1' \triangleq c_1\ \$\ n$.
- $c \triangleq c_1 \wedge_n c_2$ is equivalent to the conjunction of $c \triangleq c_1 \wedge c_2$ with $c_1 \preccurlyeq c_2'$ and $c_2 \preccurlyeq c_1'$ where $c_1' \triangleq c_1\ \$\ n$ and $c_2' \triangleq c_2\ \$\ n$.

The non-strict bounded precedence can be defined similarly (with non strict relations) as well as the operator $c_1 \vee_n c_2$. These restrictions allow us to establish the following results.

Theorem 3. Every bounded CCSL specification can be encoded **(I)** by a Boolean automata **(II)** and a PSL formula.

Proof: **(I)** We proceed by induction on the structure of CCSL specifications. As all the states in the resulting automaton are final and accepting, we do not mention them. First, let us consider CCSL relations. For every Boolean formula ϕ we denote by $\mathcal{B}_\phi$ the single-state Boolean automaton with a self loop labeled by ϕ.

- The relation $c_1 \subset c_2$ can be encoded by $\mathcal{B}_{(\overline{c}_1 \vee c_2)}$.

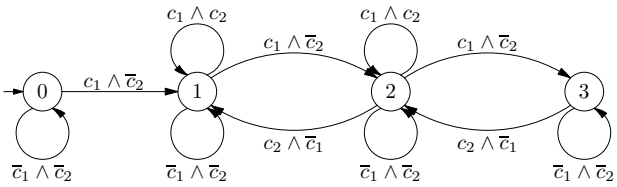

Figure 2. Boolean automaton for $c_1 \boxed{\prec_3} c_2$

• Similarly, $c_1 \boxed{\#} c_2$ can be encoded by $\mathcal{B}_{(\overline{c}_1 \vee \overline{c}_2)}$.

• The bounded precedence relation $c_1 \boxed{\prec_n} c_2$ can be encoded by an automaton with n states. These states simulate the incrementation and decrementation of a counter that stores the advance of c_1 on c_2. So one needs to move to the next state when only c_1 is true, to move back when only c_2 is true and to stay in the same state when both or neither are true (except at the bounds). Fig. 2 is the automaton for $n = 3$.

• The construction for the relation $c_1 \boxed{\preccurlyeq_n} c_2$ is similar with additional loops labeled by $c_1 \wedge c_2$ on states 0 and n and a transition from state 1 back to state 0.

A definition of the form $c \boxed{\triangleq} e$ can be encoded by the product automaton $\mathcal{A}_e \times \mathcal{B}_{c \Leftrightarrow e}$ where $\mathcal{A}_e$ is defined below.

• If e is of the form $e_1 + e_2$ then $\mathcal{A}_e$ can be obtained by making the product of $\mathcal{A}_{e_1}$, $\mathcal{A}_{e_2}$ and $\mathcal{B}_{((e_1 \vee e_2) \Leftrightarrow e)}$.

• The automaton for $e_1 * e_2$ is built similarly by replacing the third automaton by $\mathcal{B}_{((e_1 \wedge e_2) \Leftrightarrow e)}$.

• The encoding of $e_1 \searrow e_2$ is a bit more complex. Consider two copies $\mathcal{A}$ and $\mathcal{A}'$ of the product automaton $\mathcal{A}_{e_1} \times \mathcal{A}_{e_2}$. We denote by $q_0, q_1 \ldots$ the states of $\mathcal{A}$ and $q'_0, q'_1 \ldots$ the states of $\mathcal{A}'$ such that q_i and q'_i represent the same state in the different copies. We use $\mathcal{A}$ to simulate the part where e_1 has not occurred yet and $\mathcal{A}'$ the part where e_1 has occurred and we wait for the next occurrence of e_2. We have to move from $\mathcal{A}$ to $\mathcal{A}'$ when e_1 is true. Then we move back to $\mathcal{A}$ and set e to true when e_2 is true. This automaton is obtained by making the following transformations on $\mathcal{A}$ and $\mathcal{A}'$.

($\star$) For every transition $q_i \xrightarrow{\phi} q_j$ in $\mathcal{A}$ we replace the label ϕ by $\phi \wedge \overline{e}_1 \wedge \overline{e}$ and add the transition $q_i \xrightarrow{\phi \wedge e_1 \wedge \overline{e}} q'_j$ from $\mathcal{A}$ to $\mathcal{A}'$.

($\star\star$) For every transition $q'_i \xrightarrow{\phi} q'_j$ in $\mathcal{A}'$ we replace ϕ by $\phi \wedge \overline{e}_2 \wedge \overline{e}$ and add the transition $q'_i \xrightarrow{\phi \wedge e_2 \wedge e} q_j$ from $\mathcal{A}'$ to $\mathcal{A}$.

Obviously if the Boolean formula of a label reduces to false then the corresponding transition is removed (or not added).

• The encoding of $e_1 \searrow\!\!\!\!\!\!\cdot\; e_2$ is very close to the case $e_1 \searrow e_2$. The difference is that when we are in the first copy and both e_1 and e_2 are true then e is also true and we stay in the same copy. We only move to the second copy when e_1 is true and e_2 is false. So the step ($\star$) has to be replaced by:

($\star'$) For every transition $q_i \xrightarrow{\phi} q_j$ in $\mathcal{A}$ we replace the label by $\phi \wedge \overline{e}_1 \wedge \overline{e}$ and add the **two** transitions $q_i \xrightarrow{\phi \wedge e_1 \wedge \overline{e}_2 \wedge \overline{e}} q'_j$ and $q_i \xrightarrow{\phi \wedge e_1 \wedge e_2 \wedge e} q_j$.

• The encoding of $e_1 \boxed{\$_{e_2}} n$ is a generalization of the previous construction. When e_1 holds we have to wait for n positions where e_2 holds. This can be done with $n+1$ copies of $\mathcal{A}_{e_1} \times \mathcal{A}_{e_2}$. Another point of view is that a counter is encoded in the states of the resulting automaton. Similarly to the construction for the case $e_1 \searrow e_2$ we add transitions between the different copies as follows:
- from the first copy to the second when e_1 occurs,
- from the i^{th} to the $i{+}1^{\text{th}}$ when e_2 occurs for $2 \leq i \leq n{+}1$,
- from the $n+1^{\text{th}}$ to the first when e_2 occurs and this corresponds to the transitions where e must occur.

• Now we consider the filtering operation $e_1 \blacktriangledown bw$. Suppose that $bw = u \cdot v^\omega$. The expression can also be encoded similarly with $|u|{+}|v|$ copies of $\mathcal{A}_{e_1}$. Each copy is associated with positions in bw in a natural way. The transition from a copy to the next one is done when e_1 holds and after the last copy we jump to the $(|u|+1)^{\text{th}}$ one (periodic part). The variable e occurs iff e_1 occurs in a copy whose corresponding position in bw is equal to 1.

• The automaton when e is of the form $e_1 \lightning e_2$ can easily be obtained from the product automaton $\mathcal{A} = \mathcal{A}_{e_1} \times \mathcal{A}_{e_2} \times \mathcal{B}_{((e_1 \wedge \overline{e}_2) \Leftrightarrow e)}$ as follows.
- We add a sink state q_s with a loop $q_s \xrightarrow{\overline{e}} q_s$.
- We replace every transition $q \xrightarrow{\phi} q'$ in $\mathcal{A}$ $(q, q' \neq q_s)$ by $q \xrightarrow{\phi \wedge \overline{e}_2} q'$ and we add the transition $q \xrightarrow{\phi \wedge e_2} q_s$.
This operation prevents future occurrences of e as soon as e_2 has occurred.

• The way of encoding $e_1 \vee_n e_2$ is close to the bounded precedence relation since we need to store the difference between the occurrences of c_1 and c_2. To do this we need $2n+1$ states. The expression e holds when the variable that has the least occurrences holds. Fig. 3 is the automaton for $n = 2$. The right part (positive labels) corresponds to positions where the number of occurrences of c_1 is greater than c_2. So c is true in this part whenever c_2 is true. The left part is symmetrical.

• The case $e_1 \wedge_n e_2$ is quite similar. For $n = 2$ the automaton is obtained by switching e and $\overline{e}$ in the transitions that are not loops in Fig. 3.

The global automaton corresponding to a given CCSL specification is the product of all the automata corresponding to the different definitions and relations. The set of runs corresponding to such an automaton is the same as the set of models of the CCSL specification. Indeed, a careful analysis of the different steps shows that this construction strictly follows CCSL semantics.

(II) Direct consequence of **(I)** and Lemma 4. ■

The formula obtained by composing the transformation from CCSL to automata and from automata to PSL is

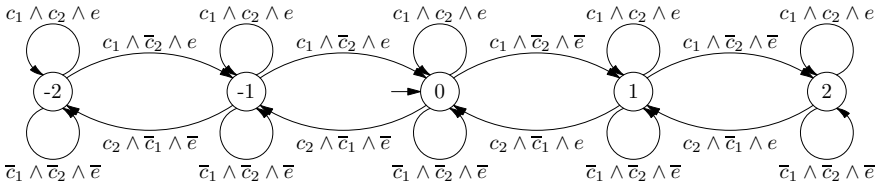

Figure 3. Boolean automaton for $c_1 \wedge_2 c_2$

not minimal. Our intent is not to define an optimal transformation but to prove that a PSL encoding is possible. Moreover, our automaton encoding should be more useful to define interfaces between CCSL and verification tools. Direct translation from bounded CCSL to PSL would not give much better results. The encoding of the counters of relations like precedence, filtering or delay is tedious with propositional variables. It should be more efficient in practice to use a richer temporal logic with counters or to consider more restricted fragments of CCSL.

In this section, we have arbitrarily chosen to bound some operators to define a syntactic fragment of CCSL. However, in there are CCSL specifications where the context already bounds the difference between the arguments of these operators. The characterization of the maximal semantic fragment of CCSL corresponding to systems with a finite state-space is an open question. Note also that the alternation operators used in the proof of Lemma 2 can be expressed in the bounded fragment. For instance, $c_1 \boxed{\overset{\sim}{=}} c_2$ is equivalent to the conjunction of $c_1 \boxed{\prec_2} c_2$ and $c_2 \boxed{\preccurlyeq_2} c'_1$ where $c'_1 \boxed{\triangleq} c_1 \$ 1$. As a consequence we have the following corollary.

Corollary 1. Every PSL formula can be encoded by a bounded CCSL specification .

V. Conclusion

We have compared the formal languages CCSL and PSL. Both languages can be used to specify behavioral properties in the domain of hardware electronic systems but at different design levels. Our results contribute to clarify their role when addressing this domain.

We have identified the CCSL constructs that cannot be expressed in PSL and the class of PSL formulas that cannot be stated in CCSL. Then, we have defined fragments of CCSL and PSL that can be encoded into each other. A sufficient condition to translate CCSL into PSL is to restrict operators that need to count the occurrences of clocks in such a way that they can be expressed with bounded counters. Conversely, CCSL cannot express the class of liveness properties but can express every PSL safety property.

We have defined translations between these fragments using an intermediate automata-based approach. The automata encoding could be used to establish comparisons of CCSL with other languages or to apply some verification algorithms. We do not claim that these preliminary results can directly be applied in concrete system design and analysis. However, this translation is an important step towards the formal verification of a CCSL specifications and the exploration of its state space.

References

[1] C. André. Syntax and semantics of the clock constraint specification language. Technical Report 6925, INRIA, 2009.

[2] C. André, F. Mallet, and J. DeAntoni. VHDL observers for clock constraint checking. In *SIES 2010*, pages 98–107, 2010.

[3] D. Bustan, D. Fisman, and J. Havlicek. Automata construction for PSL. Technical report, IBM Haifa Research Lab, 2005.

[4] R. Gascon, F. Mallet, and J. Deantoni. Logical time and temporal logics: Comparing UML MARTE/CCSL and PSL. RR 7459, INRIA, 2011. http://hal.inria.fr/inria-00540738/PDF/RR7459.pdf.

[5] R. Lazić. Safely freezing LTL. In *In FST&TCS'06*, pages 381–392. Springer, 2006.

[6] F. Mallet, C. André, and J. DeAntoni. Executing AADL models with UML/Marte. In *ICECCS'09*, pages 371–376. IEEE Computer Press, 2009.

[7] S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *TCS*, 32(3):321 – 330, 1984.

[8] OMG. *UML Profile for MARTE, v1.0*. Object Management Group, 2009. formal/2009-11-02.

[9] P. Peil, J. Medina, H. Posadas, and E. Villar. Generating heterogeneous executable specifications in SystemC from UML/MARTE models. *ISSE*, 6:65–71, 2010.

[10] IEEE standard for Property Specification Language (PSL), IEEE std 1850-2005.

[11] A. Sistla and E. Clarke. The complexity of propositional linear temporal logics. *JACM*, 32(3):733–749, 1985.

[12] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet. A co-design approach for embedded system modeling and code generation with UML and MARTE. In *DATE*, pages 226–231, 2009.