INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

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

Régis Gascon — Frédéric Mallet — Julien DeAntoni

N° 7459

December 7, 2010

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7459--FR+ENG

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

Régis Gascon , Frédéric Mallet , Julien DeAntoni

Thème COM — Systèmes communicants
Projet AOSTE

Rapport de recherche n° 7459 — December 7, 2010 — 22 pages

Abstract: The UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) provides a means to specify embedded systems. The Clock Constraint Specification Language (CCSL) allows the specification of causal, chronological and timed properties of MARTE models. Due to its purposedly 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.

We want to identify the part of MARTE/CCSL amenable to support the classical analysis methods from the Electronic Design Automation (EDA) community. In this paper, we contribute to this goal by comparing the expressiveness of CCSL and the Foundation Language of PSL. We show that none of these languages is subsumed by the other one. We identify the CCSL constructs that cannot be expressed in temporal logics and propose restrictions of 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. We define translations between these fragments of CCSL and PSL using automata as an intermediate representation.

Key-words: High-level design, Linear temporal logic, Language equivalence, Automaton based approach

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Temps logique et logiques temporelles: Comparaison de UML MARTE/CCSL et PSL

Résumé : Le profil UML MARTE (Modeling and Analysis of Real-Time and Embedded systems) permet la spécification de systèmes embarqués. Le langage associé CCSL (Clock Constraint Specification Language) est offre la possibilité de spécifier des propriétés causales, chronologiques et temporelles sur les modèles MARTE. En raison de son large spectre d'applications, CCSL a une grande expressivité qui empêche l'application de certaines techniques de vérification formelle. Cependant, la vérification formelle est une étape importante du développement dans le domaine des "hardware electronic systems". Pour ce faire, le standard IEEE PSL (Property Specification Language) fourni des notations formelles pour l'expression de propriétés en logique temporelles qui peuvent être automatiquement vérifiée sur le modèle du système électronique.

Nous voulons identifier le fragment de MARTE/CCSL susceptible de supporter les méthodes d'analyses classiques utilisées dans la communauté EDA (Electronic Design Automation). Dans ce papier, nous contribuons à ce but en comparant l'expressivité de CCSL et du fragment de PSL correspondant à la logique temporelle linéaire. Nous montrons qu'aucun de ces langages n'est inclus dans l'autre. Nous identifions les constructeurs de CCSL qui ne peuvent être exprimés par les logiques temporelles propositionnelles et proposons en conséquence des restrictions de ces opérateurs de manière à les rendre exprimable dans PSL. Réciproquement, nous identifions la classe de propriétés de PSL qui peuvent être codées dans CCSL. Nous définissons des traduction entre ces deux fragment utilisant des automates comme représentation intermédiaire.

Mots-clés : Conception haut niveau, Logique temporelle linéaire, Équivalence de langages, Approche à base d'automates.

1 Introduction

The UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE [8]) provides a mean to specify several aspects of embedded systems, ranging from large software systems on top of an operating system to specific hardware designs. The Clock Constraint Specification Language (CCSL [1]) offers a general set of notations to specify causal, chronological and timed properties on these models and has been used in various subdomains [6, 5, 2]. From this specification, it is possible to simulate the behavior of a CCSL specification at the model level. CCSL has been formally defined; however, due to its broad scope of use CCSL has an expressiveness that can prevent formal verification, since the specified system can be, by intention, non-deterministic, infinite, unbound. Very wide specifications at the system level should be progressively refined into more precise descriptions down to a point where code generation, schedulability, formal analysis become possible. MARTE/CCSL offers a support at all the refinement steps.

In the domain of hardware electronic systems, which 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).

Even though a MARTE/CCSL specification covers a broad scope and several subdomains, the intent remains to offer exhaustive verification capabilities when focusing on specific aspects within a subdomain. When focusing on hardware electronic systems, MARTE provides a support to capture structural or behavioral, functional or non-functional aspects. Its time model and CCSL , as part of MARTE, are natural candidates to express safety properties on MARTE models. 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 has been made to answer the first question [9, 13]. We are addressing here the second question.

The main contribution of this paper is then the comparison of PSL and CCSL expressiveness. The first result is that none of these languages subsume the other one. Consequently, we identified the CCSL constructs that cannot be expressed in temporal logics and proposed restrictions to these operators so that they become tractable in temporal logics. Conversely, we also identify a class of PSL formulas that can be encoded in CCSL . Then, We define translations between these fragments of CCSL and PSL using automata as an intermediate representation. These transformations make possible the combined use of both formalisms to adequately address the right level, CCSL at the model level and PSL at the implementation level. They also offer a way to provide an exhaustive analysis support for a class of CCSL specifications.

The remaining of this paper is organized as follows. In Sect. 2 we introduce CCSL and PSL and determine which kind of properties cannot be expressed in each language. We define in Sect. 3 the class of Boolean automata which is used in Sect. 4 to define translations between fragments of CCSL and PSL. Sect. 5 contains concluding remarks and future work.

2 Definitions of the languages

We define here the languages that we consider in this paper and give first comparisons related to their expressive power.

2.1 Clock Constraint Specification Language

CCSL is the companion language of MARTE UML profile for the design of embedded systems. It combines constructs from the general net theory and from the synchronous languages. CCSL offers a set of causal and timed patterns classically used in embedded systems. More formally, the language CCSL is based on the notion of *clocks* which is a general name to denote a totally ordered sequence of event occurences, 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 occur at the same time and $c_1 \boxed{\prec} c_2$ that the n^{th} occurrence of c_1 strictly precedes the n^{th} occurrence of c_2 for every $n \in \mathbb{N}^*$. The relation $c_1 \boxed{\preccurlyeq} c_2$ is the non strict version of the precedence relation.

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, a possible semantics, introduced in [1], identifies clocks with Boolean variables evolving along time . In the remaining, we will consider that 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 in the following by $|\sigma|$ the length of σ and we assume that $|\sigma| = \omega$ when σ is an infinite word. We also 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 need to know the number of occurrences of the clocks at each position of σ. We define the function χ_σ such that for every $i \in \mathbb{N}$ and $c \in \text{VAR}$ we have

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

The satisfaction of CCSL relations is defined by:

- $\sigma \models_{ccsl} c_1 \boxed{\subset} c_2$ iff for every $0 \le i \le |\sigma|$, if $c_1 \in \sigma(i)$ then $c_2 \in \sigma(i)$. We also define the coincidence relation $\boxed{=}$ such that $\sigma \models_{ccsl} c_1 \boxed{=} c_2$ 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 $0 \le i \le |\sigma|$ 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 $0 \leq i \leq |\sigma|$ such that $\chi_\sigma(c_1, i) > 0$ and $\chi_\sigma(c_2, i) > 0$ we have $\chi_\sigma(c_1, i) > \chi_\sigma(c_2, i)$.

- $\sigma \models_{ccsl} c_1 \boxed{\preccurlyeq} c_2$ iff for every $0 \leq i \leq |\sigma|$ we have $\chi_\sigma(c_1, i) \geq \chi_\sigma(c_2, i)$.

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

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

where $c \in \mathrm{VAR}$, $n \in \mathbb{N}^*$ and $bw : \mathbb{N}^* \to \mathbb{B}$ is a binary word. 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 respectively by $e_1 \quad e_2$ and $e_1 \quad e_2$. 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, $e_1 \wedge e_2$ (resp. $e_1 \vee e_2$) 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. To define this relation, we extend the function χ_σ to expressions in a natural way:

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

The satisfaction relation for expressions is defined by:

- $\sigma, i \models_{ccsl} c$ 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 $\sigma, i \models_{ccsl} e_2$.

- $\sigma, i \models_{ccsl} e_1 \quad e_2$ iff

  - $\sigma, i \models_{ccsl} e_2$,

  - there is $0 \leq j < i$ such that $\sigma, j \models_{ccsl} e_1$ and for every $j < k < i$ we have $\sigma, k \not\models_{ccsl} e_2$.

- $\sigma, i \models_{ccsl} e_1 \quad e_2$ 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 $j < k < i$ we have $\sigma, k \not\models_{ccsl} e_2$.

- $\sigma, i \models_{ccsl} e_1 \ \$_{e_2}\ n$ there is a position $0 \leq j \leq i$ such that
  - $\sigma, j \models_{ccsl} e_1$ and
  - there are exactly n distinct positions $i_1, \dots, i_n$ ($i_n = i$) such that for every $k \in \{1, \dots, n\}$ 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 $j < i$ we have $\sigma, j \not\models_{ccsl} e_2$.
- $\sigma, i \models_{ccsl} e \blacktriangledown bw$ iff
  - $\sigma, i \models_{ccsl} e$
  - $bw(\chi_\sigma(e, i)) = 1$.
- $\sigma, i \models_{ccsl} e_1 \vee e_2$ iff either
  - $\chi_\sigma(e_1, i) > \chi_\sigma(e_2, i)$ and $\sigma, i \models_{ccsl} e_1$,
  - or $\chi_\sigma(c_1, i) < \chi_\sigma(c_2, i)$ 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 $\sigma, i \models_{ccsl} e_2$.
- $\sigma, i \models_{ccsl} e_1 \wedge e_2$ 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 $\chi_\sigma(c_1, i) = \chi_\sigma(c_2, i)$ and we have $\sigma, i \models_{ccsl} e_1$ or $\sigma, i \models_{ccsl} e_2$.

A CCSL specification is a list of definitions and relations seen as a conjunction of constraints. We can represent it by a triple $\langle C, \mathit{Def}, \mathit{Rel} \rangle$ such that

- $C \subseteq$ 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 $c \boxed{\triangleq} e$ in Def we have $c \in \sigma(i)$ iff $\sigma, i \models_{ccsl} e$,

- 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 will be useful in the following:

- $c_1 - c_2$ is the difference of clocks c_1 and c_2. The definition $c \boxed{\triangleq} c_1 - c_2$ can be encoded with the definition $c_1 \boxed{\triangleq} c + c_2$ and the relation $c \boxed{\#} c_2$.
- $c \,\$_c\, n$ is a particular case of delay expression that we denote $c \,\$\, n$. This expression represents the usual synchronous delay operation. The resulting expression starts at the n^{th} occurrence of c and then coincides with c.
- Alternance relation $c_1 \boxed{\sim_{=}} c_2$ is defined by the relations $c_1 \boxed{\preccurlyeq} c_2$ and $c_2 \boxed{\prec} c_1'$ where $c_1' \boxed{\triangleq} c_1 \,\$\, 1$. Similarly, $c_1 \boxed{{}_{=}\sim} c_2$ is defined by $c_1 \boxed{\preccurlyeq} c_2$ and $c_2 \boxed{\prec} c_1'$.

2.2 Property Specification Language

The IEEE standard PSL [10] is a textual language to build temporal logic expressions. PSL assertions are used for instance in hadware design and they can be validated by model-checking or equivalence checking techniques. Compared with the classical linear temporal logic LTL, PSL provides sugaring constructs to build expressions in an easier and more concise way. However PSL is as expressive as LTL. As it would be tedious to consider the different sugaring operators of PSL in formal reasoning, we use in this paper 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 operators $\Rightarrow$ and $\Leftrightarrow$ that can be defined from the grammar above[1]. 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[2] are defined by:

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

[1] $x \Rightarrow y$ is equivalent to $\overline{x} \vee y$ and $x \Leftrightarrow y$ to $(x \Rightarrow y) \wedge (y \Rightarrow x)$.
[2] PSL standard also defines a branching time part that we do not consider here.

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.

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 some application domains of PSL such that simulation or bounded model-checking. Similarly to CCSL, the models of PSL are finite or infinite sequences over elements of 2^{VAR} that represents the set of variables that holds at each position.

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 $\models_b$ is obvious. SEREs refer to a finite (possibly empty) prefix of the model. So σ is supposed to be finite in SERE satisfaction relation (which is not the case in PSL satisfaction relation). The SERE satisfaction is defined by induction as following:

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

Finally, the satisfaction of PSL properties is defined as following.

- $\sigma \models_{psl} \neg\phi$ 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 σ_1 of σ and σ_2 such that $\sigma = \sigma_1\sigma_2$, $\sigma_1 \models_{re} r$ and $\sigma_2 \models_{psl} \phi$,
- $\sigma \models_{psl} r$ iff for every finite prefix σ_1 of σ there is a finite word σ_2 such that $\sigma_1\sigma_2 \models_{re} r \rightarrowtail \top$.

The addition of SEREs in PSL does not add expressiveness to the classical temporal logic LTL. Indeed, SEREs can be translated into LTL formulas. However, this would imply an exponential blowup of the size of the formulas.

2.3 Comparing PSL and CCSL

Since CCSL and PSL share common models, we can compare their expressive power. 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 iff $V_S \subseteq V_\phi$ and every model of ϕ is also a model of S.

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 left 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 every 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. Even with the semantics we have introduced, it is not possible to completely link the execution of a CCSL specification to a global clock. However, the states where no clocks hold are irrelevant in CCSL point of view as they do not make the system evolve. So it is not really a problem to discard them. Actually, this is what is done in the CCSL simulator TimeSquare. We will say that ϕ is simulated by S iff $V_\phi \subseteq V_S$ and every model of S *with no irrelevant states* is also a model of ϕ.

By examining the definitions PSL and CCSL, we can already make the following observations. Some CCSL relations or expressions implicitly introduce unbounded counters. For instance, one have to store the number of occurrences of the clocks c_1 and c_2 (or at least the difference between them) to encode the precedence relation $c_1 \boxed{\preccurlyeq} c_2$. 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 which is as expressive as LTL and regular expressions. The same remark holds for the expressions $c_1 \vee c_2$ and $c_1 \wedge c_2$.

On the other hand, the different CCSL relations and expressions only states safety constraints. As a specification is a conjunction of such constraints the result is always a safety property. CCSL cannot express liveness like the reachabily property $\mathsf{F}p$. A similar problem occurs for the next operator in case of finite executions. There is 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.

Lemma 2. **(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. However, we will see in the remaining of this paper that we can define large fragments of these languages that can be encoded in each other. To that aim we will first introduce a intermediate class of automata well fitted to define translations between these fragments.

3 Boolean automata

Translating directly PSL properties into CCSL is not obvious. For example, let us consider the following PSL formula:

$$\mathsf{G}(p_0 \Rightarrow \neg(\overline{p}_1 \mathsf{U} p_2)).$$

One can try to translate this property by considering its general meaning which is "there is always p_1 in an interval starting with p_0 and ending with p_2". It is more difficult to define a modular approach by composing atomic translations from PSL operator to CCSL. We use an automaton based approach. We introduce in this section a class of automata manipulating Boolean variables that we will use to establish relations between PSL and CCSL fragments.

3.1 Definition

We consider automata that handle propositional variables in VAR. The transitions of these automata are labeled by Boolean formulas interpreted like guard. 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,
- $V \subseteq$ VAR is a set of propositions,
- $\delta : Q \times \mathit{Bool}(V) \times Q$ is a transition relation where $\mathit{Bool}(V)$ is the set of Boolean formulas over VAR.

We use the definitions of Sect. 2.2 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 $\sigma : \mathbb{N} \rightarrow (Q \times 2^V)$ 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 $\mathcal{A}$ in made of the words on the alphabet 2^V corresponding to accepting runs.

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

- $Q = Q_1 \times Q_2 \times \{0,1\}$ where the last component of each state (in $\{0,1\}$) is only needed for the Büchi acceptance condition,
- $q_0 = \langle (q_0)_1, (q_0)_2, 0 \rangle$,
- $F = F_1 \times F_2 \times \{0,1\}$ and $A = Q_1 \times A_2 \times \{1\}$,
- $V = V_1 \cup V_2$,
- 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 $q_2 \xrightarrow{\phi_2} q_2'$ s.t. 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 $A_2 = Q_2$), which will be the case in the following.

3.2 CCSL and Boolean automata

Since CCSL express 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 3. Every deterministic Boolean automaton such that every execution is accepting can be simulated by a CCSL specification.

Proof. Consider a Boolean automaton $\mathcal{A} = \langle Q, I, V, \delta\rangle$. The sets of accepting and final states are not needed since every execution is accepting. So, we forget them here.

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 following:

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

where $\sum_{c\in X} c$ is the CCSL union of all the clocks in X. Similarly, we will 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:

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

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 correspond to the relations

$$(\mathbf{3}) \quad c_Q \boxed{=} Glob \qquad \text{and} \qquad q \boxed{\#} q' \text{ for every } q, q' \in Q\ (q \neq q').$$

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

$$(\mathbf{4}) \quad Glob \boxed{=} \sum_{q \in Q} I_Q.$$

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

$$(\mathbf{5}) \quad q_0 \boxed{\sim_{=}} Iq_0 \qquad \text{and} \qquad Iq \boxed{{}_{=}\!\sim} q.$$

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

We 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 $i \in \mathbb{N}$, for every $c \in V$ we have $c \in \sigma(i)$ iff $\rho(i) = \langle q_i, X_i \rangle$ and $c \in X_i$.

First we observe that for any model σ satisfying the CCSL specification, if $Iq \in \sigma(i)$ then $q \in \sigma(i+1)$ for every $i \in \mathbb{N}$. The alternance relations allows a clock q to occur only if I_q has occured between the last occurence of q and the current position (cf **(5)**). However, the definitions of the different Iq are defined w.r.t. transition relation of $\mathcal{A}$ which is deterministic and complete (cf **(2)**). This implies that exactly one Iq belongs to $\sigma(i)$ for every $i \in \mathbb{N}$. So, if $Iq \in \sigma(i)$ the only element of Q that can belonb to $\sigma(i+1)$ is q. By **(3)**, exactly one element of Q must hold at each position. This conclude the demonstration.

We proceed by induction on the position of the sequences. Suppose that we are given σ (resp. ρ). For every $i \in \mathbb{N}$ we note $\rho(i) = \langle q_i, v_i \rangle$. We show for every $i \in \mathbb{N}$ that for every position $j < i$ and variable $c \in \mathrm{VAR}$ we can build ρ (resp. σ) such that

- $c \in \sigma(j)$ iff $c \in X_j$,
- and $q_i + 1 \in \sigma(i+1)$ iff $q_i + 1$ is the state of $\rho(i+1)$.

At the begining of any model, the only clock in Q that can belong to $\sigma(0)$ is q_0. Indeed, no clock Iq has occured which prevent the other $q \in Q$ from occuring because of alternance relations (see **(5)**). Similarly, the initial sate of $\mathcal{A}$ is always q_0.

Now let σ be a model of the CCSL specification. We suppose that the property holds until position i and that we have $q_i \in \sigma(i)$ and the state of $\rho(i)$ is q_i. Since the transition relation is complete and deterministic, there is a unique $q' \in Q$ such that $q \xrightarrow{\phi} q'$ and $\sigma(i) \models \phi$. As a consequence, $Iq' \in \sigma(i)$ which implies that $q' \in \sigma(i+1)$ as we shown before. We can do the corresponding move in $\mathcal{A}$ by choosing the transition $q \xrightarrow{\phi} q'$ and setting $c \in X_i$

iff $c \in \sigma(i) \cap V$. By induction, one can build a run ρ of $\mathcal{A}$ such that for every $i \in \mathbb{N}$, for every $c \in \mathrm{VAR}$ we have $c \in \sigma(i)$ iff $c \in \rho(i)$.

Conversely, let ρ be a run of $\mathcal{A}$. We suppose that the property holds until position i, $\rho(i) = \langle q_i, v_i \rangle$ and $q_i \in \sigma(i)$. The demonstration is symmetrical. There is a unique transition $q_i \xrightarrow{\phi} q_{i+1}$ such that $v_i \models \phi$ because the transition relation is deterministic and complete. Let set $\sigma(i)$ such that for every $c \in \mathrm{VAR}$ we $c \in \sigma(i)$ iff $\rho(i) = \langle q_i, v_i \rangle$ and $v_i(c) = \top$. 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. CCSL specifications cannot be encoded by Boolean automata for the same reasons that prevent encoding CCSL specifications into PSL properties. Some relations or operators like precedence cannot be encoded by using finite state systems (see Sect. 2.3).

3.3 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 PSL formula ϕ, the construction defined in [3] can easily be adapted to build a Boolean automata accepting the set of models of ϕ. This construction itself is a slight extension of the automaton for LTL originally defined by [12]. We do not develop this construction now since the construction in the proof of upcoming Lemma 6 will follow the same main steps.

Lemma 4. From any PSL properties 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. By using the construction in [11] one can encode the behaviour of a Boolean automaton into a LTL formula.

Lemma 5. From any Boolean automaton $\mathcal{A}$, one can build a PSL formula $\phi_{\mathcal{A}}$ such that the set of models of $\phi_{\mathcal{A}}$ is exactly the set of runs of $\mathcal{A}$.

4 Translations between CCSL and PSL fragments

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

4.1 From PSL to CCSL

Lemma 3 states that 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 [4] by restricting the use of negations. A PSL formula belongs to *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. Note that one can define safety fragments of PSL by restricting temporal modalities but this one is more general. For the finite case, we also have to restrict the definition of the next operator to its weak variant (s.t. the formula is satisfied also if the model has no next position).

Lemma 6. For every property in safety PSL, one can build an automaton such that every execution is accepting.

Proof. In [3] is described a way to build automata from PSL properties. We recall below the main steps of this construction and show that the restrictions we have made allow us to obtain an automaton such that every run respecting the transition relation is accepting.

First, one can easily build a finite automaton accepting the set of finite words that corresponds to a given SERE. Indeed, SERE are essentially regular expressions. So we assume that for every SERE r there is a finite automaton $\mathcal{A}_r^f = \langle 2^{\mathrm{VAR}}, Q_r, I_r, F_r, \delta_r^f \rangle$ such that $\sigma \in \mathrm{L}(\mathcal{A}_r)$ iff $\sigma \models_{re} r$. From this automaton one can build a Büchi automaton $\mathcal{A}_r = \langle 2^{\mathrm{VAR}}, Q_r, I_r, Q_r, Q_r, \delta_r \rangle$ such that $\sigma \in \mathrm{L}(\mathcal{A}_r)$ iff $\sigma \models_{psl} r$. The transition relation δ_r is obtained by adding the following rules to δ_r^f:

$$\langle q_f, X, q_f \rangle \in \delta_r \text{ for every } q_f \in F_r \text{ and } X \in 2^{\mathrm{VAR}}.$$

This automaton has only accepting and final states. Indeed, according to PSL satisfaction relation, every prefix that can be extended to an expression satisfying the SERE must be accepted.

Then we proceed by induction on the structure of the formula. The result of the construction is an alternating automata. This allows running automata for the SERE atomic formulas in parallel of the temporal logic part. Then, it is known that an alternating Büchi automaton can be translated into a standard Büchi automaton [7].

The base case is given above. So we suppose that for every subformula ψ of ϕ we can build an automaton $\mathcal{A}_\psi = \langle 2^{\mathrm{VAR}}, Q_\psi, I_\psi, A_\psi, F_\psi, \delta_r \rangle$ such that such that every run is accepting and $\sigma \in \mathrm{L}(\mathcal{A}_\psi)$ iff $\sigma \models_{psl} \psi$. There are actually two constructions because the case where the formula ϕ is of the form $\neg(r \rightarrowtail \psi)$ must be treated separately. In that case, $\mathcal{A}_\phi$ is built from the finite automaton $\mathcal{A}_r^f = \langle 2^{\mathrm{VAR}}, Q_r, I_r, F_r, \delta_r^f \rangle$ and $\mathcal{A}_\psi = \langle 2^{\mathrm{VAR}}, Q_\psi, I_\psi, Q_\psi, Q_\psi, \delta_\psi \rangle$ as follows.

- the set of states is the union of Q_r and Q_ψ and an additional state q_t,
- the set of initial states is I_r,
- the set of final states is the union of q_t, Q_r and $F_{\neg\psi}$, so $F_\phi = Q_\phi$ because $F_{\neg\psi} = Q_{\neg\psi}$,
- the set of accepting states is the union of q_t, Q_r and $A_{\neg\psi}$, so $A_\phi = Q_\phi$ because $A_{\neg\psi} = Q_{\neg\psi}$,

- For every $q_f \in F_r$ and $X \in 2^{\mathrm{VAR}}$ we have

$$\delta(q_f, X) = \bigwedge_{q' \in \delta_r(q,X)} q' \wedge \delta_{\neg\psi}(q_0, X)$$

  where q_0 is the initial state of $\mathcal{A}_{\neg\psi}$.

- For every $q \in Q_r \setminus F_r$ and $X \in 2^{\mathrm{VAR}}$

  - if $\delta_r(q, X)$ is defined then

$$\delta(q, X) = \bigwedge_{q' \in \delta_r(q,X)} q'$$

    ,

  - otherwise $\delta(q, X) = q_t$,

- for every $q \in Q_{\neg\psi}$ the transition relation coincides with $\delta_{\neg\psi}$,

- finally $\delta(q_t, X) = q_t$ for every $X \in 2^{\mathrm{VAR}}$.

Note that we only have to consider negated occurrences of $r \rightarrowtail \psi$ by definition of the safety fragment.

For the other cases, $\mathcal{A}$ is defined as follows. The set of states is composed of

- the set of states of the automata $\mathcal{A}_r$ for every SERE r occurring in ϕ,
- the set of states of the automata $\mathcal{A}_{\neg(r \rightarrowtail \psi)}$ for every subformula $r \rightarrowtail \psi$ occurring in ϕ,
- the set of subformulas of ϕ and their negation (we identify $\neg\neg\psi$ with ψ).

The initial state is ϕ. The transition relation δ is defined recursively:

- $\delta(p, X) = \top$ iff $p \in X$.
- $\delta(r, X) = \delta(q_0^r, X)$ where q_0^r is the initial state of $\mathcal{A}_r$.
- $\delta(\phi_1 \wedge \phi_2, X) = \delta(\phi_1, X) \wedge \delta(\phi_2, X)$.
- $\delta(\phi_1 \vee \phi_2, X) = \delta(\phi_1, X) \vee \delta(\phi_2, X)$.
- $\delta(\neg\phi, X) = \overline{\delta(\phi, X)}$.
- $\delta(\mathsf{X}\phi, X) = \phi$.
- $\delta(\phi_1 \mathsf{U} \phi_2, X) = \delta(\phi_2, X) \vee (\delta(\phi_1, X) \wedge \phi_1 \mathsf{U} \phi_2)$.

Where the overlined expressions are interpreted as follows:

- $\overline{a \wedge b} = \overline{a} \vee \overline{b}$,
- $\overline{a \vee b} = \overline{a} \wedge \overline{b}$,
- $\overline{\delta(q_0^{\neg r}, X)} = \delta(q_0^r, X)$ where q_0^r is the initial state of the automaton $\mathcal{A}_r$,
- $\overline{\delta(q_0^{r \rightarrowtail \psi}, X)} = \delta(q_0^{\neg(r \rightarrowtail \psi)}, X)$ where $q_0^{\neg(r \rightarrowtail \psi)}$ is the initial state of $\mathcal{A}_{\neg(r \rightarrowtail \psi)}$,
- $\overline{\psi} = \neg\psi$, for every subformula (we still identify $\neg\neg\psi$ with ψ).

Note that because we consider the safety fragment the cases $\overline{\delta(q_0^r, X)}$ and $\overline{\delta(q_0^{r \rightarrowtail \psi}, X)}$ never occur (see restrictions of negation).

In the general construction, the accepting states would be those of the form $\neg\phi_1 \mathsf{U} \phi_2$ or the states of the automata $\mathcal{A}_{\neg(r \rightarrowtail \psi)}$ and $\mathcal{A}_r$. However, for formulas in the safety fragment the construction above is particular. For every run of the automaton obtained it cannot be the case that an infinite branch does not encounter one of those final states.

We proceed by induction. If we are in a state of the form p, the branch is finite. Similarly, in the cases r or $\neg(r \rightarrowtail \psi)$ we are done since all the states of the corresponding automata are accepting. Now we suppose that every subformula of ϕ and its negation satisfy the property. In almost all cases, the different branches of $\delta(\phi, X)$ goes to states labeled by strict subformulas of ϕ. In that cases we can use the induction hypothesis to conclude. The only remaining case is when ϕ is of the form $\phi_1 \mathsf{U} \phi_2$ or $\neg\phi_1 \mathsf{U} \phi_2$. The first case cannot arise since we are in the safety fragment. In the second case the transition rule is the follows:

$$\delta(\neg\phi_1 \mathsf{U} \phi_2) = \overline{\delta(\phi_1 \mathsf{U} \phi_2)} = \delta(\neg\phi_2, X) \wedge (\delta(\neg\phi_1, X) \vee \neg(\phi_1 \mathsf{U} \phi_2)).$$

We can use the induction hypothesis on the branch corresponding to ϕ_2. For the other branch, we can prove by induction that

- either we reach a position when $\neg\phi_2$ and $\neg\phi_1$ hold and then we can use the induction hypothesis,
- or $\neg(\phi_1 \mathsf{U} \phi_2)$ is visited infinitely often. Since this state is accepting we are also done.

So we can set $A_\phi = F_\phi = Q_\phi$.

By construction, every state of the alternating Büchi automaton obtained is final and accepting. If we use the powerset construction of [7] to build an equivalent non-alternating automaton, the sets of final and accepting states are also equal to the whole set of states. So every run of the resulting automaton is accepting. □

The proof is given in Appendix **??** is a variant of the construction in [3]. We just have to ensure that every execution is accepting. By Lemmas 6 and 3 we can encode every safety PSL formula into CCSL specifications.

Lemma 7. Every safety PSL formula can be encoded by a CCSL specification.

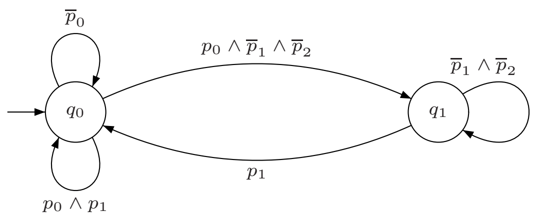
$\overline{p}_0$
$p_0 \wedge \overline{p}_1 \wedge \overline{p}_2$
$\overline{p}_1 \wedge \overline{p}_2$
q_0
q_1
p_1
$p_0 \wedge p_1$

Figure 1: Boolean automaton for $\mathsf{G}(p_0 \Rightarrow \neg(\overline{p}_1 \mathsf{U} p_2))$

For instance, Figure 1 represents the automaton corresponding to the formula $\mathsf{G}(p_0 \Rightarrow \neg(\overline{p}_1 \mathsf{U} p_2))$ after simplifications. This automaton corresponds to the CCSL specification $\langle V, \mathit{Def}, \mathit{Rel}\rangle$ such that $V = \{p_0, p_1, p_2\}$ and

$$\mathit{Def} = \left\{ \begin{array}{c} Q \boxed{\triangleq} q_0 + q_1 \quad , \quad \mathit{Glob} \boxed{\triangleq} Q + p_0 + p_1 + p_2, \\ Iq_0 \boxed{\triangleq} ((q_0 - p_0) + (q_0 * p_0 * p_1) + (q_1 * p_1)), \\ Iq_1 \boxed{\triangleq} (((q_0 * p_0) - (p_1 + p_2)) + (q_1 - (p_0 + p_1))) \end{array} \right\},$$

$$\mathit{Rel} = \left\{ \begin{array}{c} \mathit{Glob} \boxed{=} Q \quad , \quad q_0 \boxed{\#} q_1 \quad , \quad \mathit{Glob} \boxed{=} Iq_0 + Iq_1, \\ q_0 \boxed{\underset{=}{\sim}} Iq_0 \quad , \quad Iq_1 \boxed{\overset{\sim}{=}} q_1 \end{array} \right\}.$$

4.2 From CCSL to PSL

To obtain a fragment of CCSL that can be encoded in PSL, we restrict the precedence relations and the operators $c_1 \vee c_2$ and $c_1 \wedge c_2$. We define a precedence relation such that the advance of the fastest clock is bounded. We denote these relations $\boxed{\prec_n}$ and $\boxed{\preccurlyeq_n}$ where $n \in \mathbb{N}$. A model σ satisfies $c_1 \boxed{\prec_n} c_2$ iff for every $i \in \mathbb{N}$ we have $\chi_\sigma(c_2, i) < \chi_\sigma(c_1, i) \leq \chi(c_2, i) + m$. The relation $\boxed{\preccurlyeq_n}$ is defined similarly with non strict inequalities. We define similar variants $c_1 \boxed{\wedge_n} c_2$ and $c_1 \boxed{\vee_n} c_2$ that restrict the difference of the clocks c_1 and c_2 to be bounded by n. Such expressions are particular since they also imposes implicit constraints on the parameters. However, this is the most convenient way of defining a syntactic fragment of CCSL that can be translated into CCSL.

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 a fragment of CCSL. Indeed, the operators can be defined in full CCSL:

- $c_1 \boxed{\prec_n} c_2$ is equivalent to $c_1 \boxed{\prec} c_2$ and $c_2 \boxed{\preccurlyeq} c_1'$ where $c_1' \boxed{\triangleq} c_1 \$ n$. Non strict precedence case is similar.

- $c \boxed{\triangleq} c_1 \boxed{\wedge_n} c_2$ is equivalent to the conjunction of $c \boxed{\triangleq} c_1 \vee c_2$ with the relations $c_1 \boxed{\preccurlyeq} c_2'$ and $c_2 \boxed{\preccurlyeq} c_1'$ where $c_1' \boxed{\triangleq} c_1 \$ n$ and $c_2' \boxed{\triangleq} c_2 \$ n$. This equivalence make clear the relations implicitly imposed on the parameters of the expression. The operator $c_1 \boxed{\vee_n} c_2$ can be defined similarly.

These restrictions allow us to establish the following results.

Lemma 8. **(I)** Every bounded CCSL specification can be encoded by a Boolean automata. **(II)** Every bounded CCSL specifications can be encoded by a PSL formula.

Proof. **(I)** We proceed by induction on the structure of CCSL specifications. As every state in the resulting automata 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 subclocking relation $c_1 \boxed{\subset} c_2$ can be encoded by $\mathcal{B}_{(\overline{c}_1 \vee c_2)}$.

- Similarly, the exclusion relation $c_1 \boxed{\#} c_2$ 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 store the advance of c_1 on c_2. So one needs to move to the next state when c_1 is true, to move back when c_2 is true and to stay in the same state when both (or none) are true. Fig. 2 is the automaton for $n = 3$.

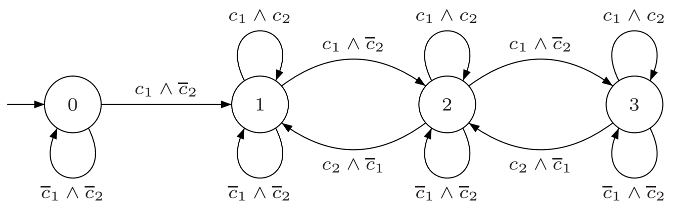
0
1
2
3
$c_1 \wedge \overline{c}_2$
$c_1 \wedge c_2$
$c_1 \wedge \overline{c}_2$
$c_1 \wedge c_2$
$c_1 \wedge \overline{c}_2$
$c_1 \wedge c_2$
$c_2 \wedge \overline{c}_1$
$c_2 \wedge \overline{c}_1$
$\overline{c}_1 \wedge \overline{c}_2$
$\overline{c}_1 \wedge \overline{c}_2$
$\overline{c}_1 \wedge \overline{c}_2$
$\overline{c}_1 \wedge \overline{c}_2$

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

- The construction for the relation $c_1 \boxed{\preccurlyeq_n} c_2$ is similar an additional loop labeled $c_1 \wedge c_2$ on state 0 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 $e_1 \quad e_2$ 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 $q_0, q_1 \dots$ the states of $\mathcal{A}$ and $q'_0, q'_1 \dots$ the states of $\mathcal{A}'$ such that q_i and q'_i represent the same state in the different copies.

  To build the automaton $\mathcal{A}_e$ 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. So, 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 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 the label 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 $e_1 \quad e_2$ is very close to the case $e_1 \quad e_2$. 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

  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 $e_1 \, \$_{e_2} \, n$ 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. In the same way than the construction for the case $e_1 \quad e_2$ we add transitions between the different copy as following:

  – 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 following.
  - 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 a e_2 has occured.

- The way of encoding $e_1 \boxed{\wedge_n} e_2$ 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 here $2n+1$ states. The expression e holds when the variable that has the less 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 when c_2 is true. The left part is symmetrical.

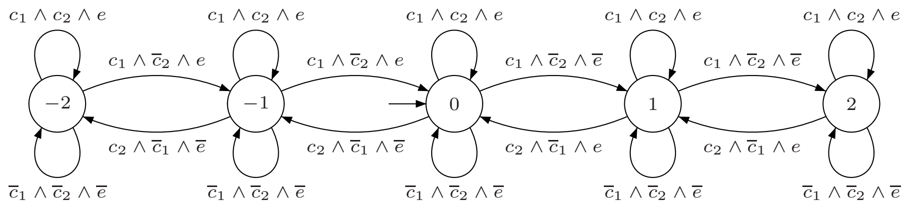
$c_1 \wedge c_2 \wedge e$
$c_1 \wedge \overline{c}_2 \wedge e$
$c_1 \wedge \overline{c}_2 \wedge \overline{e}$
-2
-1
0
1
2
$c_2 \wedge \overline{c}_1 \wedge \overline{e}$
$c_2 \wedge \overline{c}_1 \wedge e$
$\overline{c}_1 \wedge \overline{c}_2 \wedge \overline{e}$

Figure 3: Boolean automaton for $c_1 \boxed{\wedge_2} c_2$

- The case $e_1 \boxed{\vee_n} e_2$ 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 models corresponding to such an automaton is the same than the set of models of the CCSL specification. A careful analysis of the different steps shows that this construction strictly follows CCSL semantics.

(II) This second part is a direct consequence of **(I)** and Lemma 5, even if the PSL formula obtained by composing the two transformations is not minimal. We can define a direct translation from bounded CCSL to PSL. However, the result of the translation remains complicated. We still have to encode the counters of relations like precedence, filtering, delay. . . which is tedious when using only propositional variables. □

Here we have arbitrarily chosen to bound the precedence operators. There are examples where the context already bounds the difference between the arguments of a precedence relation (see for instance the definition of alternance in Sect. 2.1). So, bounded CCSL is not the largest fragment that can be encoded in PSL. Determining whether the state space of a CCSL specification is finite is an open question. Moreover it seems very difficult to determine a syntactic fragment corresponding to such CCSL specifications.

5 Conclusion

In this paper, we have compared the expressiveness of CCSL and PSL, two formal languages used for similare purposes but at different levels. We have identified the CCSL constructs that cannot be expressed in PSL and the class of PSL formulas that cannot be stated in CCSL . We have also defined the common fragments between CCSL and PSL so that one can be translated into the other. A sufficient condition to translate CCSL specifications into PSL is to bound the integer counters used to count the number of occurrences of clocks. Precisely, the relative advance of the clocks put in relation by these CCSL constructs must be bounded. This translation is an important step towards the formal verification of a CCSL specification and the exploration of its state space. In the future, we can also take benefits of the intermediate translation to automata to establish comparison with other languages.

Conversely, we have defined the translation of PSL safety properties into CCSL. CCSL cannot express the class of liveness properties. CCSL has not been designed for this purpose. However, it can be interesting to capture all the expressive power of PSL in a higher level description language. A solution to fill the gap could be to introduce temporal modalities in a CCSL -like language while keeping the multi-clocks aspects. CCSL is indeed a language that is still evolving. We are currently defining a minimal kernel from which all the relations and expressions introduced in this paper (and possibly others) can be derived. In this development, we should maintain the correspondences with the other languages involved in system design such as PSL.

References

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

[2] C. André, F. Mallet, and J. DeAntoni. VHDL observers for clock constraint checking. In *Industrial Embedded Systems (SIES), 2010 International Symposium on*, pages 98–107, July 2010.

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

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

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

[6] F. Mallet, M.-A. Peraldi-Frati, and C. André. Marte CCSL to execute East-ADL timing requirements. In *ISORC'09*, pages 249–253, Japan, Tokyo, March 2009. IEEE Computer Press.

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

[8] OMG. *UML Profile for MARTE, v1.0.* Object Management Group, November 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. *Innovations in Systems and Software Engineering*, 6:65–71, 2010. 10.1007/s11334-009-0105-4.

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

[11] A. Sistla and E. Clarke. The complexity of propositional linear temporal logic. *J. ACM*, 32(3):733–74, 1985.

[12] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS'86*, pages 332–344. IEEE, 1986.

[13] 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.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399