INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Verification of clock constraints: CCSL Observers in Esterel

Charles André

N° 7211

February 2010

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--7211--FR+ENG

Verification of clock constraints: CCSL Observers in Esterel

Charles André*

Thème COM — Systèmes communicants
Projet Aoste

Rapport de recherche n° 7211 — February 2010 — 59 pages

Abstract: The Clock Constraint Specification Language (CCSL) has been informally introduced in the specifications of the UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE). In a previous report entitled "Syntax and Semantics of the Clock Constraint Specification Language", we equipped a kernel of CCSL with an operational semantics. In the present report we pursue this clarification effort by giving a mathematical characterization to each CCSL constructs. We also propose a systematic approach to the formal verification of CCSL constraints with dedicated Observers. A comprehensive library of Esterel modules, which supports this approach, is provided.

Key-words: CCSL, UML, time constraints, verification, observer, Esterel

* Université de Nice Sophia Antipolis

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

Vérification de contraintes d'horloges : Observateurs CCSL en Estérel

Résumé : Le langage de Spécification de Contraintes d'Horloges (connu sous le nom de Clock Constraint Specification Language ou sous l'acronyme CCSL) a été introduit de façon informelle dans le document de spécification du profil UML pour la modélisation et l'analyse des systèmes temps réel et embarqués (MARTE). Dans un précédent rapport intitulé "Syntax and Semantics of the Clock Constraint Specification Language" nous avons défini une sémantique opérationnelle pour un noyau de CCSL. Le présent rapport poursuit cet effort de formalisation en donnant une caractérisation mathématique précise à chaque élément du langage CCSL. Nous proposons également une approche systématique pour la vérification formelle de contraintes CCSL s'appuyant sur le concept d'Observateur. Une bibliothèque complète de modules Esterel est fournie pour la mise en œuvre de cette approche.

Mots-clés : CCSL, UML, contraintes temporelles, vérification, observateurs, Esterel

1 Introduction

Modeling of distributed systems, as well as electronic systems with multi-cores or multi-clock domains, needs multiple time bases. The UML profile for *Modeling and Analysis of Real-Time and Embedded* systems [1] (MARTE) addresses this modeling issue through its rich *model of Time*. MARTE also introduces the concept of *clock constraints* and proposes the non-normative language CCSL (short for *Clock Constraint Specification Language*) for specifying such constraints.

The MARTE time model has been first presented at the 10^{th} international conference on *Model Driven Engineering Languages and Systems* [2]. This paper introduced the concepts of time bases, *clocks* and clock constraints. CCSL appeared only in a few illustrative examples. On the other hand, the OMG specification of MARTE contains neither a precise syntax nor a formal semantics for CCSL; only an informal (English) semantics is given. This lack of formal description of CCSL has been partially filled in our research report entitled *"Syntax and Semantics of the Clock Constraint Specification Language"* [3], which described a kernel of CCSL and provided a structural operational semantics for this kernel. The present report contributes further to the semantics of CCSL by giving full mathematical characterization of the clock relations and clock expressions of CCSL.

CCSL has also been used in *verification of time requirements* [4, 5]. The first reference proposed simulation of CCSL specifications to find constraint violations. The second one dealt with formal verification: we first specified time constraints of a simplified signal processing application with CCSL, and we then used *model checking* techniques to verify that an Esterel program effectively met these specifications. Esterel [6, 7] is an imperative synchronous language, well-suited to control-dominated system programming. As usual with synchronous languages [8, 9], we applied an *observer*-based approach [10] for safety property verification. The second contribution of the present report is to propose a general approach to CCSL constraint checking with CCSL-observers.

The report consists of three main sections. The first section introduces the *multiform logical* time, the *clocks*, and the *clock constraints*. A clock constraint is a *clock relation* that applies to two *clock expressions*. In this section, all the clock relations from the kernel CCSL, and two often used derived relations (*alternation* and *synchronization*), are mathematically defined. The primitive clock expressions of the kernel CCSL, and some derived expressions are characterized in the same way.

The second section first explains the concept of *observer*. Then, a general approach is proposed for the implementation of (CCSL) *observers*, *generators*, and *adaptors*, which are modules that implement clock relations, clock expressions, and CCSL clocks, respectively. The behavior of each module is dictated by the operational semantics given in the previous report on CCSL [3]. For each relation, the condition under which the relation is *violated* is clearly stated.

The third section describes, in details, an Esterel module library library that implements the proposed verification approach.

2 Clock Constraint Specification Language

2.1 Multiform logical time

MARTE Time model deals with both discrete and dense time. In MARTE, a *clock* gives access to a *time structure* made of *time bases*, which are themselves ordered sets of *instants*. A clock can be either *chronometric* or *logical*. The former is related to "physical time" while the latter is not. This report focuses on the structural relations between clocks and these relations do not differentiate between chronometric and logical clocks. However, some relations only apply to discrete clocks (logical or chronometric) and others apply to both discrete and dense clocks. Dense clocks considered in MARTE are necessarily chronometric. Logical clocks refer to discrete-time logical clocks and represent *logical time*.

Leslie Lamport [11] introduced logical clocks in the late 70's. The logical clocks associate numbers (logical timestamps) with events in a distributed system, such that there exists a consistent total ordering of all the events of the system. These clocks can be implemented by counters with no actual timing mechanisms. In the 80's, the synchronous languages [12] introduced their own concept of logical time. This logical time shares with Lamport's time the fact that they need not actually refer to physical time. Logical time only relies on (partial or total) ordering of instants. In what follows, we consider logical time in the sense of synchronous languages. In the synchronous language Signal, a *signal* s is an infinite totally ordered sequence $(s_t)_{t\in\mathbb{N}}$ of typed elements. Index t denotes a *logical instant*. At each logical instant of *its* clock, a signal is present and carries a unique value. Signal is a multi-clock (or polychronous) language: it does not assume the existence of a *global clock*. Instead, it allows multiple logical clocks. Signal composition is ruled by operators which are either mono-clock operators (composing signals defined on a same clock) or multi-clock operators (allowing composition of signals having different clocks).

Indeed, a logical clock can be associated with any event. This point of view has been adopted in the MARTE time model [1, Chap. 10]. A logical clock "ticks" with each new occurrence of its associated event. Synchronous languages like Esterel exploit this property. In an Esterel program, time may be counted in seconds, meters, laps... (see the Berry's RUNNER program [7] which describes the training of a runner). This variety of events supporting time leads to the concept of *multiform time*. More technical examples can be found in automotive applications. For instance, the electronic ignition is driven by the angular position of the crankshaft rather than by a chronometric time (see our study of a knock controller in a 4-stroke engine [13]).

In this report, we consider only (discrete) logical clocks and their relationships through *clock constraints*. Many aspects are extendable to dense clocks, but these are not addressed in this report.

2.2 Clocks

In the MARTE time model, a *clock* is a model element giving access to the time structure underlying the application. A clock refers to a time base. In this report, we adopt a simplified

model, already used in our papers (see for instance [14]). This model hides the concept of time base and considers that a clock directly owns an ordered set of instants.

A *Clock* is a 5-tuple $\langle \mathcal{I}, \prec, \mathcal{D}, \lambda, u \rangle$ where $\mathcal{I}$ is a set of instants, $\prec$ is a quasi-order relation on $\mathcal{I}$, named *strict precedence*, $\mathcal{D}$ is a set of labels, $\lambda : \mathcal{I} \to \mathcal{D}$ is a labeling function, u is a symbol, standing for a *unit*. For logical clocks, u is often called tick, it can be processorCycle as well or any other logical activation of a behavior. The *ordered set* $\langle \mathcal{I}, \prec \rangle$ is the temporal structure associated with the clock.$\prec$ is a total, irreflexive, and transitive binary relation on $\mathcal{I}$.

A *discrete-time clock* is a clock with a discrete set of instants $\mathcal{I}$. Since $\mathcal{I}$ is discrete, it can be indexed by natural numbers in a fashion that respects the ordering on $\mathcal{I}$: let $\mathbb{N}^\star = \mathbb{N} \setminus \{0\}$, $\mathrm{idx} : \mathcal{I} \to \mathbb{N}^\star$, $\forall i \in \mathcal{I}$, $\mathrm{idx}(i) = k$ if and only if i is the k^{th} instant in $\mathcal{I}$.

For any discrete time clock $c = \langle \mathcal{I}_c, \prec_c, \mathcal{D}_c, \lambda_c, u_c \rangle$, $c[k]$ denotes the k^{th} instant in $\mathcal{I}_c$ (*i.e.*, $k = \mathrm{idx}_c(c[k])$). For any instant $i \in \mathcal{I}_c$ of a discrete time clock, $^\circ i$ is the unique immediate predecessor of i in $\mathcal{I}_c$. For simplicity, we assume a virtual instant the index of which is 0, and which is the (virtual) immediate predecessor of the first instant. i° is the unique immediate successor of i in $\mathcal{I}_c$, if any.

2.2.1 Time structure

A *Time Structure* is a pair $\langle C, \preccurlyeq \rangle$ where C is a set of clocks, $\preccurlyeq$ is a binary relation on $\bigcup_{c \in C} \mathcal{I}_c$, named *precedence*. $\preccurlyeq$ is reflexive and transitive. From $\preccurlyeq$ we derive four new relations:

- *Coincidence* ($\equiv \triangleq \preccurlyeq \cap \preccurlyeq^{-1}$),
- *Strict precedence* ($\prec \triangleq \preccurlyeq \setminus \equiv$),
- *Independence* ($\| \triangleq \overline{\preccurlyeq \cup \preccurlyeq^{-1}}$), and
- *Exclusion* ($\# \triangleq \prec \cup \prec^{-1}$).

The graphical representation of instant relations is given in Table 1.

instant relation	symbol	graphical representation
strict precedence	$\prec$	
(non strict) precedence	$\preccurlyeq$	
coincidence	$\equiv$	
exclusion	#	

Table 1: Instant relations

Let $I = \left(\bigcup_{c \in C} \mathcal{I}_c\right) / \equiv$ (the set of instants quotiented by the equivalence relation $\equiv$). The Time Structure $T = \langle C, \preccurlyeq \rangle$ is *well-structured* if $\langle I, \preccurlyeq \rangle$ is a partially ordered set (POset).

2.3 Clock constraints

Specifying a full time structure using only instant relations is not realistic. Moreover a set of instants is usually infinite, thus forbidding an enumerative specification of instant relations. Hence the idea to extend relations to clocks. We have introduced the concept of *clock constraints* in the MARTE specification (chapter 10) and also a dedicated language for expressing such constraints: CCSL [1, Annex C.3]. This language is non normative (the MARTE profile implementors are not obliged to support it). The semantics of CCSL given in the specification is informal. A first formal semantics, based on mathematical expressions has been proposed in a paper [14] and a research report [15], which is an extended version of the paper. A precise definition of the syntax of a *kernel* of CCSL along with a structural operational semantics is now available [3, 16]. This semantics is the golden reference for the CCSL constraint solver implemented in TimeSquare[1], the software environment that supports CCSL and the MARTE time profile.

Clock constraints are classified into four categories:

1. coincidence-based constraints (also known as synchronous constraints),
2. precedence-based constraints (also known as asynchronous constraints),
3. mixed constraints, which combine synchronous and asynchronous aspects,
4. NFP chronometric constraints (NFP stands for Non Functional Property).

The last category, which is pertinent for chronometric clocks only, is not considered in this report.

A CCSL specification $\mathcal{S}$ consists of *clock declarations* and a set of binary *clock relations*. These relations apply to clocks or *clock expressions*. A *run* (execution sequence) represents a legal evolution. Each *step* of the run consists of a set of clocks that are fired[2]. Of course, all the clocks fired during a step must respect $\mathcal{S}$. For each clock, at any step of a run, the operational semantics of CCSL [3] allows the computation of an enabling condition (necessary condition for the clock to fire) and the change in state induced by its possible firing.

The basic (*i.e.,* part of the kernel) clock relations, some usual derived clock relations, and a set of clock expressions are presented in the next subsections. Most of these definitions are borrowed from the previously mentioned papers on CCSL.

2.4 Clock relations

2.4.1 Primitive clock relations

Let a and b two discrete time clocks. Five primitive relations on clocks are defined:

[1] `http://www-sop.inria.fr/aoste/dev/time_square`

[2] "A clock fires" or "a clock ticks" are two equivalent expressions equally used in this report.

- *Subclocking:* $a \boxed{\subset} b$ is a synchronous clock relation. There exists a mapping h from $\mathcal{I}_a$ to $\mathcal{I}_b$ which is injective and order preserving. a is said to be a sub-clock of b, and b a super-clock of a.

$$\begin{aligned} a \boxed{\subset} b \Leftrightarrow & \\ & \Big((\forall k \in \mathbb{N}^\star, a[k] \in \mathcal{I}_a)(\exists l \in \mathbb{N}^\star, b[l] \in \mathcal{I}_b)(a[k] \equiv b[l] = h(a[k]))\Big) \wedge \\ & \Big((\forall k_1, k_2 \in \mathbb{N}^\star, a[k_1], a[k_2] \in \mathcal{I}_a)(a[k_1] \prec a[k_2] \Rightarrow h(a[k_1]) \prec h(a[k_2]))\Big) \end{aligned} \tag{1}$$

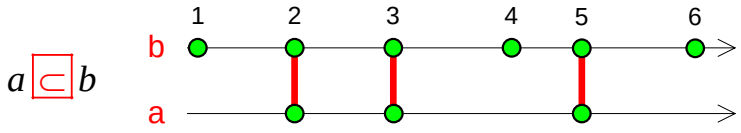


Figure 1: Exemple of subclocking.

- *Tight Subclocking:* $a \boxed{\Subset} b$ is a subclocking relation in which the image of $\mathcal{I}_a$ by h is an interval of $\mathcal{I}_b$. In a formal way:

$$\begin{aligned} a \boxed{\Subset} b \Leftrightarrow & \\ & \Big((\exists j \in \mathbb{N})(\forall k \in \mathbb{N}^\star, a[k] \in \mathcal{I}_a)((b[j+k] \in \mathcal{I}_b) \wedge (a[k] \equiv b[j+k]))\Big) \wedge \\ & \Big((\forall k_1, k_2 \in \mathbb{N}^\star, a[k_1], a[k_2] \in \mathcal{I}_a)(a[k_1] \prec a[k_2] \Rightarrow h(a[k_1]) \prec h(a[k_2]))\Big) \end{aligned} \tag{2}$$

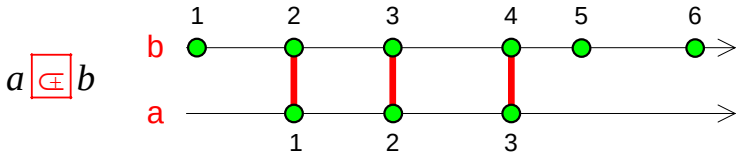


Figure 2: Exemple of tight subclocking.

In this example clock a is finite ($|\mathcal{I}_a| = 3$) and $j = 1$.

- *Strict precedence:* $a \boxed{\prec} b$ is an asynchronous clock relation. a is said to be strictly faster than b, and b strictly slower than a.

$$a \boxed{\prec} b \Leftrightarrow (\forall k \in \mathbb{N}^\star, b[k] \in \mathcal{I}_b)((a[k] \in \mathcal{I}_a) \wedge (a[k] \prec b[k])) \tag{3}$$

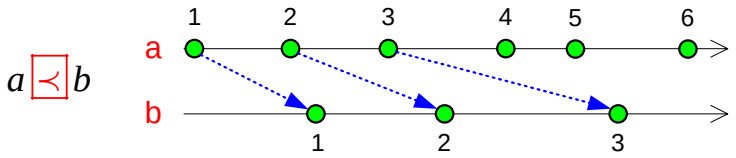


Figure 3: Exemple of strict precedence.

- *Precedence:* $a \boxed{\preccurlyeq} b$ is similar to the previous one but considering the non strict precedence instead. a is said to be faster than b, and b slower than a.

$$a \boxed{\preccurlyeq} b \Leftrightarrow \big(\forall k \in \mathbb{N}^\star, b[k] \in \mathcal{I}_b\big)\big((a[k] \in \mathcal{I}_a) \wedge (a[k] \preccurlyeq b[k])\big) \tag{4}$$

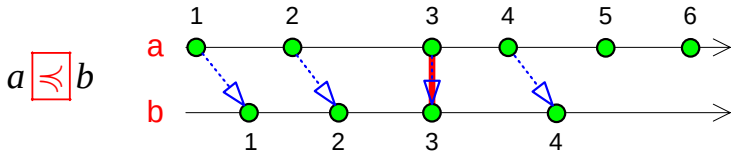


Figure 4: Exemple of non strict precedence.

In this example, note that $a[3]$ and $b[3]$ are coincident.

- *Exclusion:* $a \boxed{\#} b$ means that a and b have no coincident instants.

$$a \boxed{\#} b \Leftrightarrow \big(\forall j \in \mathbb{N}^\star, a[j] \in \mathcal{I}_a\big)\big(\forall k \in \mathbb{N}^\star, b[k] \in \mathcal{I}_b\big)\big(\neg(a[j] \equiv b[k])\big) \tag{5}$$

2.4.2 Derived clock relations

We mention three often used derived clock relations: Equality, Alternation, Synchronization.

Equality $a \boxed{=} b$ is a typical synchronous clock relation derived from tight subclocking.

$$a \boxed{=} b \Leftrightarrow \big(a \boxed{\Subset} b\big) \wedge \big(b \boxed{\Subset} a\big) \tag{6}$$

Hence, there is a bijection between instants of a and b. This bijection is order preserving and the instants are point-wise coincident: $\forall k \in \mathbb{N}^\star, a[k] \equiv b[k]$. Hence another characterization of the clock equality relation is given in equation 7.

$$\begin{aligned} a \boxed{=} b \Leftrightarrow \\ &\Big(\big(\forall k \in \mathbb{N}^\star, a[k] \in \mathcal{I}_a\big)\big(b[k] \in \mathcal{I}_b\big) \wedge \big(a[k] \equiv b[k]\big)\Big) \wedge \\ &\Big(\big(\forall k \in \mathbb{N}^\star, b[k] \in \mathcal{I}_b\big)\big(a[k] \in \mathcal{I}_a\big) \wedge \big(a[k] \equiv b[k]\big)\Big) \end{aligned} \tag{7}$$

For infinite sets, the specification simplifies to equation 8:

$$a \boxed{=} b \Leftrightarrow \big((\forall k \in \mathbb{N}^\star)(a[k] \equiv b[k])\big) \tag{8}$$

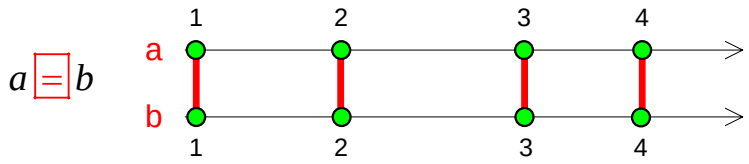


Figure 5: Exemple of clock equality.

The next two relations are asynchronous. They involve precedence relations and auxiliary clocks. Given a clock c, we denote $c\$1$ the tight sub-clock of c such that

$$\big(\forall k \in \mathbb{N}^\star, c[k+1] \in \mathcal{I}_c\big)\big(c\$1[k] \in \mathcal{I}_{c\$1}\big) \wedge \big(c\$1[k] \equiv c[k+1]\big) \tag{9}$$

In other words, $c\$1$ is c deprived of its first instant. We will see later on page 18, how this can be easily defined with clock expressions.

Alternation $a \boxed{\sim} b$. The strict form of alternation can be specified as

$$a \boxed{\sim} b \Leftrightarrow \big(a \boxed{\prec} b\big) \mid \big(b \boxed{\prec} a\$1\big) \tag{10}$$

The composition of clock relation (operator $\mid$) imposes the conjunction of the relations. So, the alternation can be also defined as

$$a \boxed{\sim} b \Leftrightarrow \big(\forall k \in \mathbb{N}^\star, a[k+1] \in \mathcal{I}_a\big)\big((b[k] \in \mathcal{I}_b) \wedge (a[k] \prec b[k] \prec a[k+1])\big) \tag{11}$$

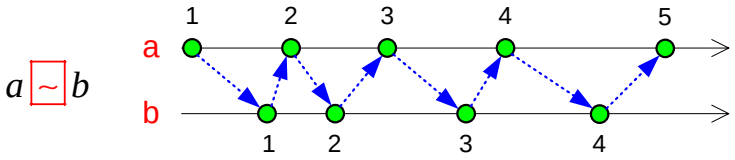


Figure 6: Exemple of strict alternation.

This definition is far simpler when only infinite sets are considered:

$$a \boxed{\sim} b \Leftrightarrow \big(\forall k \in \mathbb{N}^\star\big)\big(a[k] \prec b[k] \prec a[k+1]\big) \tag{12}$$

For the sake of simplicity, from now on, we give only definitions for infinite sets. Since there exists a non strict form of precedence ($\boxed{\preccurlyeq}$), three other variants of alternation can be defined.

- RNS-alternation (right non-strict alternation)

$$a \boxed{\sim_{=}} b \Leftrightarrow \big(\forall k \in \mathbb{N}^{\star}\big)\big(a[k] \preccurlyeq b[k] \prec a[k+1]\big) \tag{13}$$

- LNS-alternation (left non-strict alternation)

$$a \boxed{{}_{=}\sim} b \Leftrightarrow \big(\forall k \in \mathbb{N}^{\star}\big)\big(a[k] \prec b[k] \preccurlyeq a[k+1]\big) \tag{14}$$

- NS-alternation (non-strict alternation, which is both right and left non-strict)

$$a \boxed{{}_{=}\sim_{=}} b \Leftrightarrow \big(\forall k \in \mathbb{N}^{\star}\big)\big(a[k] \preccurlyeq b[k] \preccurlyeq a[k+1]\big) \tag{15}$$

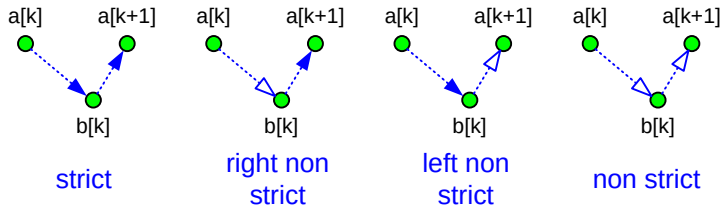


Figure 7: Various forms of alternation.

Synchronization This relation, unlike the alternation, does not impose a strict ordering on instants of a and b. Instead, the k^{th} instants of a and b are not ordered, but they both precede the $k+1^{th}$ instants of a and b. Figure 8 shows an example of strict synchronization. This relation is borrowed from the General Net Theory and stands for a *synchronic distance* of 2 (see Reisig's book [17], chapter 4).

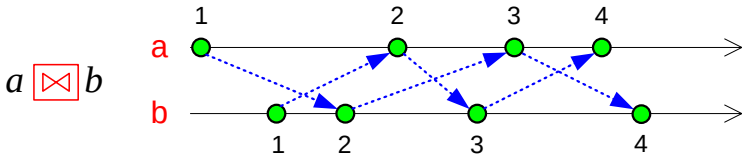


Figure 8: Exemple of strict synchronization.

Like the alternation, synchronization has four variants:

$$a \boxed{\bowtie} b \Leftrightarrow \Big((a \boxed{\prec} b\$1) \mid (b \boxed{\prec} a\$1) \Big) \tag{16}$$

$$a \boxed{\bowtie_{=}} b \Leftrightarrow \Big((a \boxed{\preccurlyeq} b\$1) \mid (b \boxed{\prec} a\$1) \Big) \tag{17}$$

$$a \boxed{{}_{=}\bowtie} b \Leftrightarrow \Big((a \boxed{\prec} b\$1) \mid (b \boxed{\preccurlyeq} a\$1) \Big) \tag{18}$$

$$a \boxed{{}_{=}\bowtie_{=}} b \Leftrightarrow \Big((a \boxed{\preccurlyeq} b\$1) \mid (b \boxed{\preccurlyeq} a\$1) \Big) \tag{19}$$

Using quantifiers, the infinite forms can be redefined as follows:

- synchronization (strict synchronization)

$$a \boxed{\bowtie} b \Leftrightarrow \big(\forall k \in \mathbb{N}^{\star}\big)\big((a[k] \prec b[k+1]) \wedge (b[k] \prec a[k+1])\big) \tag{20}$$

- RNS-synchronization (right non-strict synchronization)

$$a \boxed{\bowtie_{=}} b \Leftrightarrow \big(\forall k \in \mathbb{N}^{\star}\big)\big((a[k] \preccurlyeq b[k+1]) \wedge (b[k] \prec a[k+1])\big) \tag{21}$$

- LNS-synchronization (left non-strict synchronization)

$$a \boxed{{}_{=}\bowtie} b \Leftrightarrow \big(\forall k \in \mathbb{N}^{\star}\big)\big((a[k] \prec b[k+1]) \wedge (b[k] \preccurlyeq a[k+1])\big) \tag{22}$$

- NS-synchronization (non-strict synchronization, which is both right and left non-strict)

$$a \boxed{{}_{=}\bowtie_{=}} b \Leftrightarrow \big(\forall k \in \mathbb{N}^{\star}\big)\big((a[k] \preccurlyeq b[k+1]) \wedge (b[k] \preccurlyeq a[k+1])\big) \tag{23}$$

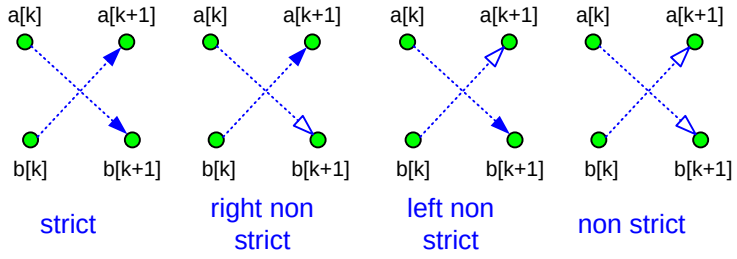


Figure 9: Various forms of synchronization.

2.5 Clock expressions

A clock expression defines a new implicit clock. In this subsection we name this implicit clock c.

2.5.1 Index independent clock expressions

Clock union $a + b$ defines a clock that ticks whenever a or b ticks.

$$\begin{array}{l}\text{Let } c \boxed{=} a + b \text{ the following properties hold:}\\ \quad 1)\ \big(a \boxed{\subset} c\big) \wedge\\ \quad 2)\ \big(b \boxed{\subset} c\big) \wedge\\ \quad 3)\ \Big(\big(\forall i \in \mathcal{I}_c\big)\big(\exists j \in \mathcal{I}_a \cup \mathcal{I}_b\big) i \equiv j\Big)\end{array} \tag{24}$$

Equations 24-1 and 24-2 state that c is a super-clock of both a and b. Equation 24-3 makes c minimal with respect to the subclocking relation.

Clock intersection $a * b$ defines a clock that ticks whenever both a or b tick.

$$\begin{array}{l}\text{Let } c \boxed{=} a * b \text{ the following properties hold:}\\ \quad 1)\ \big(c \boxed{\subset} a\big) \wedge\\ \quad 2)\ \big(c \boxed{\subset} b\big) \wedge\\ \quad 3)\ \Big(\big(\forall i \in \mathcal{I}_a\big)\big(\forall j \in \mathcal{I}_b\big)\big(\exists k \in \mathcal{I}_c\big)(i \equiv j) \Rightarrow (i \equiv k)\Big)\end{array} \tag{25}$$

Equations 25-1 and 25-2 state that c is a sub-clock of both a and b. Equation 25-3 makes c maximal with respect to the subclocking relation.

Clock difference $a - b$ defines a clock that ticks whenever a ticks but b does not.

$$\begin{array}{l}\text{Let } c \boxed{=} a - b \text{ the following properties hold:}\\ \quad 1)\ \big(c \boxed{\subset} a\big) \wedge\\ \quad 2)\ \Big(\big(\forall i \in \mathcal{I}_a, \nexists j \in \mathcal{I}_b, i \equiv j\big)\big(\exists k \in \mathcal{I}_c\big) i \equiv k\Big)\end{array} \tag{26}$$

Equation 26-1 states that c is a sub-clock of a. Equation 26-2 makes c maximal with respect to the subclocking relation by ensuring that all the instants of a not coincident with an instant of b have a coinciding instant in c.

Figure 10 illustrates the three clock expressions just defined.

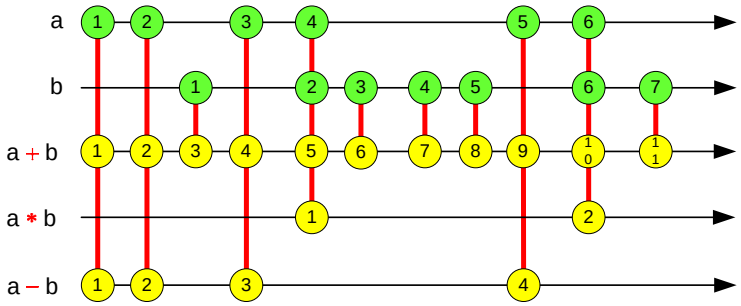


Figure 10: Exemples of clock expressions union, intersection, and difference.

2.5.2 Index dependent clock expressions

As from $a \upharpoonright k$, where k is a natural number, defines a clock that is a tight sub-clock of a starting after index k.

$$c \boxed{=} a \upharpoonright k \Leftrightarrow \left(c \boxed{\in} a\right) \wedge \left(c[1] \equiv a[k+1]\right) \tag{27}$$

This clock expression is illustrated in figure 11. The virtual intial instant of $a \upharpoonright k$, indicates that the *birth* of this clock is between instant k and $k+1$ of clock a.

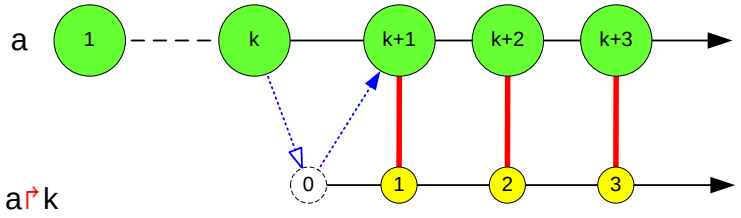


Figure 11: Exemples of clock expression 'as from'.

A variant of this expression is $a \upharpoonright *$. It is dynamically evaluated during a *run* and takes $\chi(a)$ as its parameter.

Sup $a \vee b$ defines a clock that is the fastest among all the clocks slower than a and b (equation 28).

$$\text{Let } \mathcal{C}^{\preccurlyeq}_{a,b} = \left\{ d \in \mathcal{C} \mid \left(a \boxed{\preccurlyeq} d\right) \wedge \left(b \boxed{\preccurlyeq} d\right) \right\},\ \left(\forall c' \in \mathcal{C}^{\preccurlyeq}_{a,b}\right)\left((a \vee b) \boxed{\preccurlyeq} c'\right) \tag{28}$$

This fixpoint relation can be also expressed using instants:

$$\begin{array}{l}\text{Let } c \boxed{=} a \vee b \text{ the following properties hold:}\\ \quad (\forall k \in \mathbb{N}^\star, c[k] \in \mathcal{I}_c)\\ \quad\quad 1)\ \left(a[k] \in \mathcal{I}_a\right) \wedge\\ \quad\quad 2)\ \left(b[k] \in \mathcal{I}_b\right) \wedge\\ \quad\quad 3)\ \left(c[k] \equiv \begin{cases} a[k] \text{ if } b[k] \preccurlyeq a[k] \\ b[k] \text{ if } a[k] \preccurlyeq b[k] \end{cases}\right)\end{array} \tag{29}$$

Equations 29-1 and 29-2 state that for any instant $c[k]$ of c, there exist instants with the same index in both a and b. Equation 29-3 imposes that $c[k]$ is coincident with the later of $a[k]$ and $b[k]$. Note that the two branches of the case-statement (equation 29-3) are not exclusive. There may be the case that $c[k] \equiv a[k] \equiv b[k]$.

Inf $a \wedge b$ defines a clock that is the slowest among all the clocks faster than a and b (equation 30).

$$\text{Let } \mathcal{C}^{\succcurlyeq}_{a,b} = \left\{ d \in \mathcal{C} \mid \left(d \boxed{\preccurlyeq} a\right) \wedge \left(d \boxed{\preccurlyeq} b\right) \right\},\ \left(\forall c' \in \mathcal{C}^{\succcurlyeq}_{a,b}\right)\left(c' \boxed{\preccurlyeq} (a \wedge b)\right) \tag{30}$$

This fixpoint relation can be also expressed using instants:

$$\begin{array}{l}\text{Let } c \boxed{=} a \wedge b \text{ the following properties hold:}\\ \quad (\forall k \in \mathbb{N}^\star, c[k] \in \mathcal{I}_c)\\ \quad\quad 1)\ \left(\left(a[k] \in \mathcal{I}_a\right) \wedge \left(b[k] \in \mathcal{I}_b \Rightarrow a[k] \preccurlyeq b[k]\right) \wedge \left(c[k] \equiv a[k]\right)\right) \vee\\ \quad\quad 2)\ \left(\left(b[k] \in \mathcal{I}_b\right) \wedge \left(a[k] \in \mathcal{I}_a \Rightarrow b[k] \preccurlyeq a[k]\right) \wedge \left(c[k] \equiv b[k]\right)\right)\end{array} \tag{31}$$

Equation 31-1 imposes that any instant $c[k]$ of c coincides with instant $a[k]$ when $a[k]$ precedes $b[k]$ or $b[k]$ does not exist. Equation 31-2 imposes that $c[k]$ coincides with instant $b[k]$ when $b[k]$ precedes $a[k]$ or $a[k]$ does not exist. Note that if $a[k] \equiv b[k]$, the two equations 31-1 and 31-2 are not exclusive. In this case, $c[k] \equiv a[k] \equiv b[k]$. Figure 12 shows the sup and inf of two clocks.

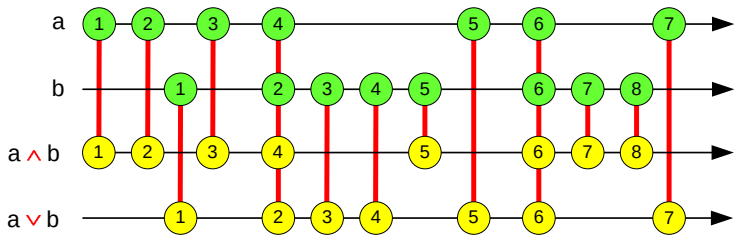


Figure 12: Exemples of clock expressions sup and inf.

2.5.3 Clock expressions with clock death

Upto $a \lightning b$ defines a clock c that ticks whenever a ticks upto the first tick of b. As of this tick, c dies (can not tick any more).

$$\begin{aligned}\text{Let } c \boxed{=} a \lightning b \text{ the following properties hold:}\\ 1)\ \big(c \boxed{\subset} a\big) \wedge \\ 2)\ \big((\forall k \in \mathbb{N}^\star, a[k] \in \mathcal{I}_a)(a[k] \prec b[1]) \Rightarrow (c[k] \equiv a[k])\big)\end{aligned} \tag{32}$$

Equation 32-1 imposes c to be a sub-clock of a. Equation 32-2 specifies that any instant of a that strictly precedes the first instant of b, is coincident with an instant of c. Figure 13 is an illustration where $|\mathcal{I}_a| = 3$.

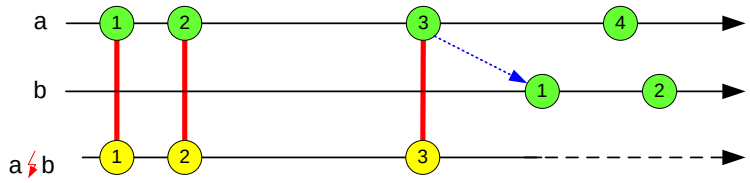


Figure 13: Exemples of clock expression upto.

The clock expression *awaiting*, which also dies, is defined on page 15.

2.5.4 Clock expressions with schedule

Here, schedule has the meaning of a plan that gives a list of events or tasks and the times at which each one should happen or be done. More precisely, the schedule associated with an expression contains the future times at which the clock should tick in coincidence with a given clock. Clock expressions like *sampling* have a very short schedule that contains at most one future coincidence. Others, like *defer* or *filtering* may be arbitrary large.

Awaiting $a \hat{} n$, where $n \in \mathbb{N}^\star$, is a synchronous clock expression. This expression waits for the n^{th} strictly future tick of a. On this occurrence, a ticks once and dies.

$$\begin{aligned}\text{Let } c \boxed{=} a \hat{} n \text{ the following properties hold:}\\ 1)\ \big(|\mathcal{I}_c| = 1\big) \wedge \\ 2)\ \big((\exists a[n] \in \mathcal{I}_a)(c[1] \equiv a[n])\big)\end{aligned} \tag{33}$$

Note that $a \hat{} n$ ticks once and dies (see figure 14).

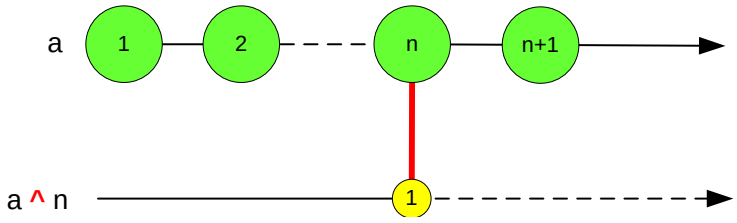


Figure 14: Exemple of clock awaiting.

Strict sampling $a \searrow b$ is a mixed clock expression (*i.e.*, based on both precedence and coincidence). It defines a subclock of b that ticks whenever clock a has ticked at least once since the previous tick of b. Figure 15 shows the corresponding instant ordering.

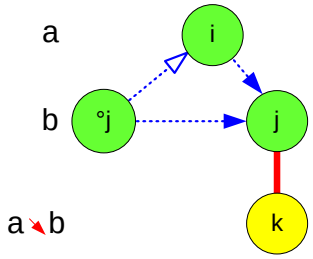


Figure 15: Strict sampling instant ordering.

Let $c \boxed{=} a \searrow b$ the following properties hold:

$$\begin{aligned} &1)\ \left(c \boxed{\subset} b\right) \wedge \\ &2)\ \left((\forall i \in \mathcal{I}_a)(\exists j \in \mathcal{I}_b, {}^{\circ}j \preccurlyeq i \prec j) \Rightarrow (\exists k \in \mathcal{I}_c, j \equiv k)\right) \end{aligned} \tag{34}$$

Equation 34-1 says that c is a subclock of b. Equation 34-2 specifies the ordering relations given in figure 15.

Non strict sampling $a \searrow b$ is also a mixed clock expression, similar to the previous one, just changing the precedence relations. Figure 16 shows the corresponding instant ordering.

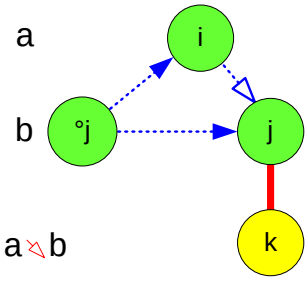


Figure 16: Non strict sampling instant ordering.

Let $c \boxed{=} a \searrow b$ the following properties hold:

$$\begin{aligned} &1)\ \left(c \boxed{\subset} b\right) \wedge \\ &2)\ \left((\forall i \in \mathcal{I}_a)(\exists j \in \mathcal{I}_b, {}^\circ j \prec i \preccurlyeq j) \Rightarrow (\exists k \in \mathcal{I}_c, j \equiv k)\right) \end{aligned} \tag{35}$$

Figure 17 highlights the different behaviors of the strict and non strict clock expression samplings.

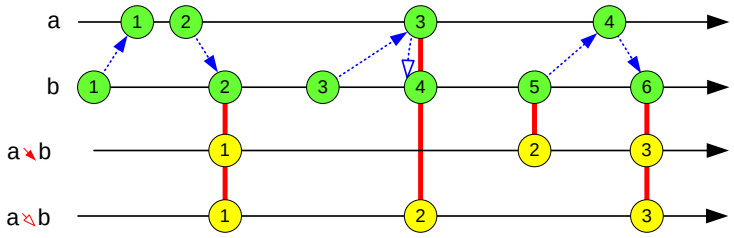


Figure 17: Exemple of clock sampling.

Defer $a\ (\ n\ ) \rightsquigarrow b$ is also a mixed clock expression that deals with multiple future scheduled ticks. Figure 18 shows the corresponding instant ordering.

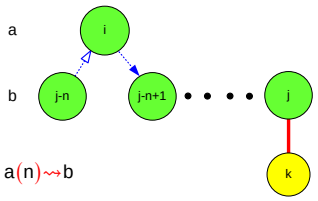


Figure 18: Defer instant ordering.

Let $c \boxed{=} a\ (\ n\ )\ b$ the following properties hold:

$$\begin{array}{ll} 1) & \left(c \boxed{\subset} b\right) \wedge \\ 2) & \big((\forall i \in \mathbb{N}^\star, a[i] \in \mathcal{I}_a)(\exists n_i \in \mathbb{N}^\star) \\ & (\exists j \in \mathbb{N}^\star, j \geqslant n_i, b[j] \in \mathcal{I}_b, b[j-n_i] \preccurlyeq a[i] \prec b[j-n_i+1]) \\ & (\exists k \in \mathbb{N}^\star, c[k] \in \mathcal{I}_c, c[k] \equiv b[j])\big) \end{array} \tag{36}$$

Figure 19 shows clock expression 'defer' when n is a constant. In this case, the clock expression is also known as clock expression 'delayedFor'.

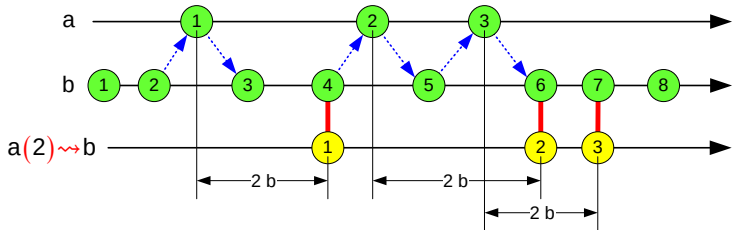


Figure 19: Exemple of clock expression 'defer'.

Filtering $a \blacktriangledown w$, where w is a binary word is a often used synchronous clock expression. Binary words are described in Annex A.

$$a \blacktriangledown w \Leftrightarrow \left((\forall k \in \mathbb{N}^\star, c[k] \in \mathcal{I}_c)(c[k] \equiv a[w \uparrow k])\right) \tag{37}$$

$w \uparrow k$ denotes the index of the k^{th} '1' in the binary word w. So, c is a sub-clock of a. Example in figure 20 considers a periodic binary word (transient part '01', periodic part '100'). Note that the operation \$1 defined in equation 9 might have been defined by filtering, using the periodic binary word $0\,(1)^\omega$.

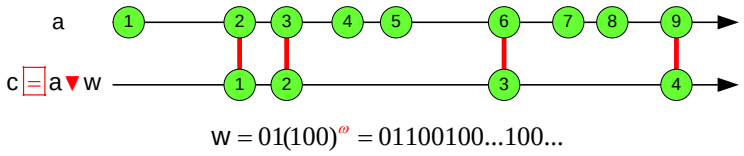


Figure 20: Exemple of filtering by a binary word.

2.5.5 Clock expression concatenation

Concat $a \bullet b$ defines a clock c that ticks whenever a ticks upto the death of a. As of this tick, c ticks whenever b ticks.

Let $c \boxed{=} a \bullet b$ the following properties hold:

$$\begin{aligned} &\text{Let } l = |\mathcal{I}_a|,\ \big(\forall k \in \mathbb{N}^\star, c[k] \in \mathcal{I}_c\big) \\ &\quad 1)\ \big((k \leqslant l) \wedge (c[k] \equiv a[k])\big)\ \vee \\ &\quad 2)\ \big((k > l) \wedge (\exists m \in \mathbb{N}^\star, b[m] \preccurlyeq a[l] \prec b[m+1])\ \wedge (b[k+m-l] \in \mathcal{I}_b) \\ &\qquad \wedge\ (c[k] \equiv b[m+k-l])\big) \end{aligned} \tag{38}$$

Equation 38-1 deals with the case when a is alive, whereas equation 38-2 addresses the case when a is dead. Figure 21 is an illustration where $l = |\mathcal{I}_a| = 5$ and $m = 3$.

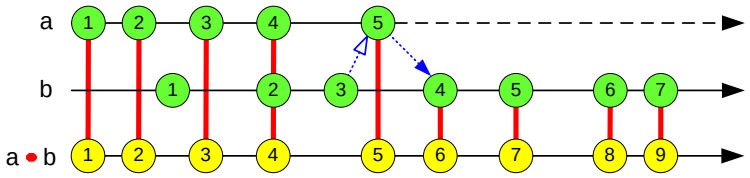


Figure 21: Exemples of clock expression concat.

3 Observers

3.1 Verification by observers

Verification by observers is a technique widely applied to synchronous languages [10]. The principle is given in figure 22. An *observer* (right-hand box in the figure) is a reactive program expressing a safety property P that has to be verified for a program (middle box). In synchronous language, the observer is put in (synchronous) parallel with the program. The observer has a unique output that signals possible violations of P. The observer receives the same input signals as the program. It also receives its output signals. Thus, the observer is purely passive: it only listen to the program without interfering with it. Often, a property holds only under some contexts. The *assumptions* made on the system environment are represented by another reactive program called *Environment* (left box in the figure). The Environment only generates useful input sequences.

The verification consists of checking that the synchronous parallel composition of the three reactive programs Environment, Program, and Observer never emits a violation for any input sequence provided by the Environment. The analysis can be done by standard reachability analysis techniques. If the property is false, an input sequence leading to the violation can be generated. This is called a counter-example.

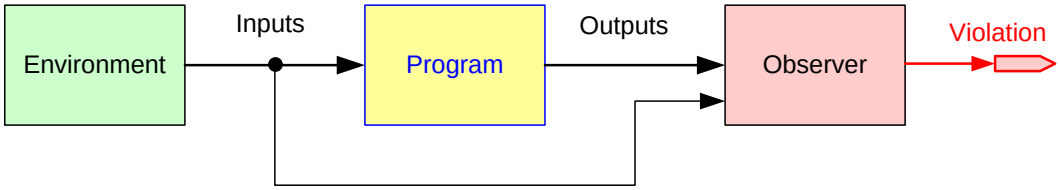


Figure 22: Property checking of reactive programs.

With the synchronous languages, the observers can be written in the very same language as the program to verify. This is illustrated with the language Esterel. Before, we give a guide-line for implementing CCSL observers.

3.2 Principle of the implementation

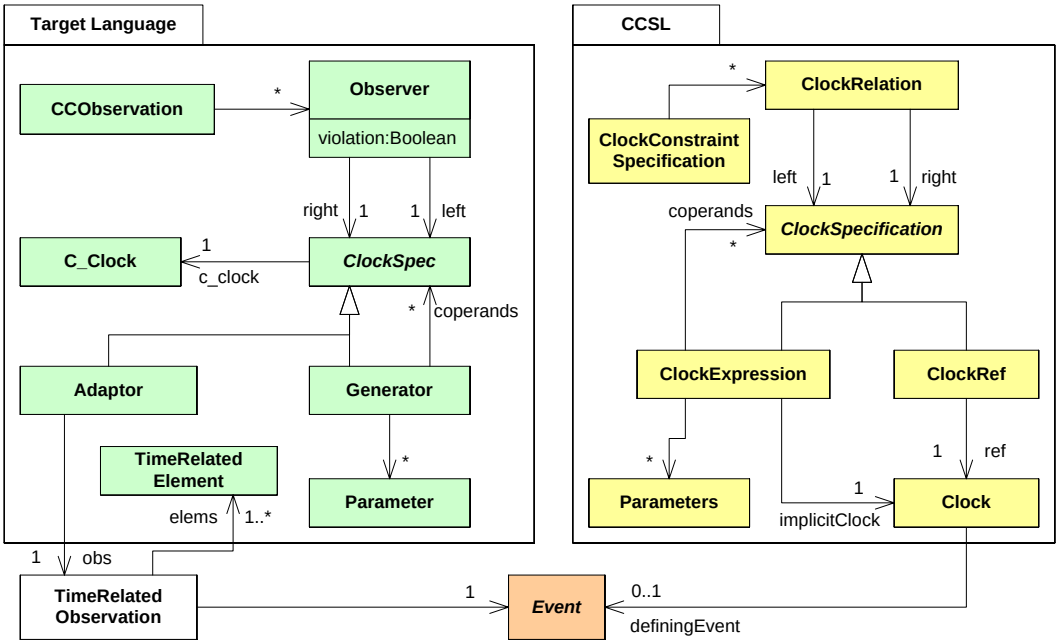


Figure 23: Metamodel for property observation.

We are interested in temporal property checking. These properties are expressed in CCSL and rely on the concept of logical clocks. On its left side, figure 23 contains a metamodel for temporal property observations. The simplified metamodel of CCSL has been put on

the right-hand side. The correspondance between the metamodel elements is summarized in table 2. In our approach, we adopt a uniform naming convention for the identifiers of programming elements related to time property observations. The third column gives these prefixes.

Table 2: Metamodel element correspondances.

CCSL	Target Language	Prefix
ClockConstraintSpecification	CCObservation	
ClockRelation	Observer	`Ccsl_R_`
ClockExpression	Generator	`Ccsl_E_`
	Adaptor	`Ccsl_A_`
Clock	C_Clock	`c_`

Conceptually, a logical clock may represent an *event* whose occurrences are represented by the instants of the clock. An *adaptor* yields a C_Clock, representation of a CCSL clock in the target language. To determine whether a c_clock has to tick or not, the adoptor observes the state of one or several objects[3] of the program (property elems in the metamodel). For instance, in VHDL, a rising edge of a signal a, while a signal b is set to '1' (statement "a'event **and** a='1' **and** b='1'") may represent an event and thus be associated with a c_clock. A *generator* represents a CCSL clock expression. Like an adaptor, a generator has a c_clock as output. Finally, an *observer* represents a CCSL clock relation. In the target language implementation an observer provides a Boolean called violation.

We propose a modular approach. For a given target language (Esterel in this report), we provides a library of modules for observers, generators, and adaptors. Figure 24 illustrates the use of such a library for property verification. The next three sub-sections describe what should be the behavior of the various observers, generators, and adaptors, independently of any target language. The behavior results from the operational semantics of CCSL described in the report on the *kernel* CCSL [3]. This semantics transforms any CCSL specification $\mathcal{S}$ into a Boolean expression $[\![\mathcal{S}]\!]$. These Boolean expressions use extended logical operators defined in table 3.

Operator	Meaning	Equivalent expression
$i \Rightarrow j$	implication	$\neg i \vee j$
$i = j$	equality	$(i \wedge j) \vee (\neg i \wedge \neg j)$
$i \# j$	exclusion	$\neg(i \wedge j)$
$i \oplus j$	exclusive disjunction	$(\neg i \wedge j) \vee (i \wedge \neg j)$

Table 3: Extended logical operators

[3]Here, object has not the restrictive meaning given by object languages. An object can be a variable (*e.g.*, in C), a signal (*e.g.*, Esterel, VHDL), etc.

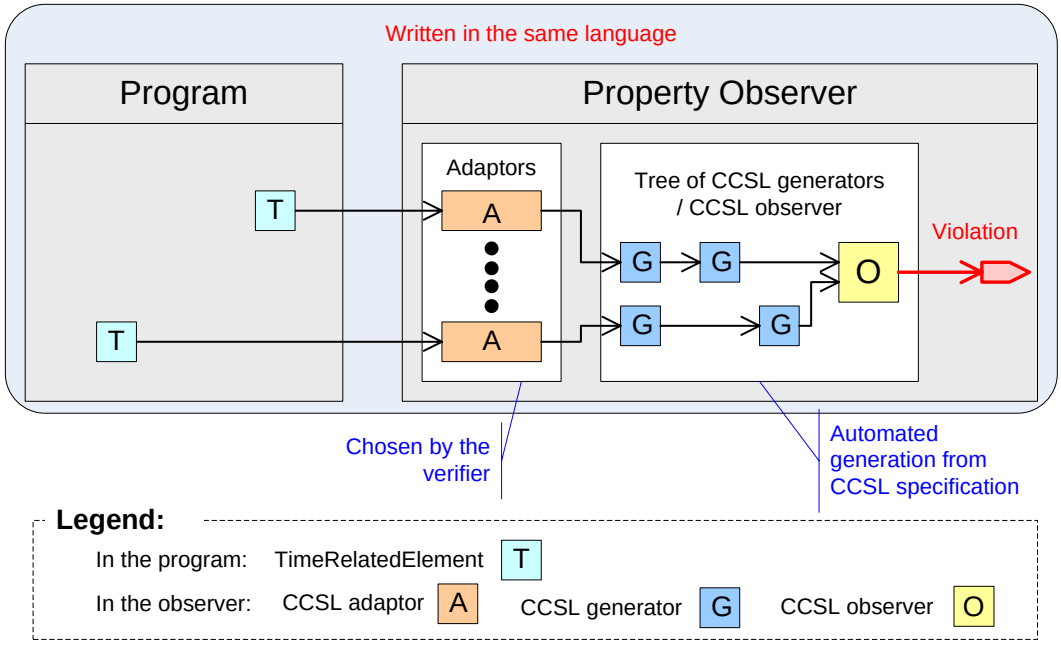


Figure 24: Use of CCSL observer libraries.

For each clock c, $\chi(c)$ is the index of its current instant in a run, and $[\![c]\!]$ is a Boolean variable associated with c. When this variable is true, the associated clock can be fired at the current step.

3.3 Observers of clock relations

3.3.1 Observers for primitive clock relations

Table 4 groups together semantic information relative to the primitive clock relations. The first column indicates the relation, the second column the enabling condition, the third column the possible changes in state induced by clock firings. The last column gives the logical expression for the *violation* of the clock relation. This expression is the logical negation of the condition contained in the second column. An observer must evaluate the violation condition and update an internal state, if needed.

Table 4: Violations of primitive clock relations

Relation	Enabling	Changes	Violation
$a \subset b$	$[\![a]\!] \Rightarrow [\![b]\!]$		$[\![a]\!] \wedge \neg [\![b]\!]$

Continued on next page

Relation	Enabling	Changes	Violation
$a \boxed{\Subset} b$	$[\![a]\!] = [\![b]\!]$		$[\![a]\!] \oplus [\![b]\!]$
$a \boxed{\#} b$	$[\![a]\!] \# [\![b]\!]$		$[\![a]\!] \wedge [\![b]\!]$
$a \boxed{\prec} b$	$(\delta = 0) \Rightarrow \neg [\![b]\!]$	$\delta \leftarrow \begin{cases} \delta + 1 & \text{if } [\![a]\!] \wedge \neg [\![b]\!] \\ \delta - 1 & \text{if } \neg [\![a]\!] \wedge [\![b]\!] \\ \delta & \text{otherwise.} \end{cases}$	$(\delta = 0) \wedge [\![b]\!]$
$a \boxed{\preccurlyeq} b$	$(\delta = 0) \Rightarrow ([\![b]\!] \Rightarrow [\![a]\!])$	$\delta \leftarrow \begin{cases} \delta + 1 & \text{if } [\![a]\!] \wedge \neg [\![b]\!] \\ \delta - 1 & \text{if } \neg [\![a]\!] \wedge [\![b]\!] \\ \delta & \text{otherwise.} \end{cases}$	$(\delta = 0) \wedge [\![b]\!] \wedge \neg [\![a]\!]$

where $\delta \triangleq \chi(a) - \chi(b)$

3.3.2 Observers for derived clock relations

Equality At any step of a run, for any existing clocks a and b, the clock relation $a \boxed{=} b$ is the same as the tight subclocking relation. The condition and the validation are the same as those of the tight subclocking (see table 5).

Alternation This relation is not primitive. Its enabling and violation conditions result from a composition of the enabling and violation conditions of the composed relations.

Starting with the definition $a \boxed{\sim} b \Leftrightarrow (a \boxed{\prec} b) \mid (b \boxed{\prec} a\$1)$, we apply the transformations rules to each precedence. This yields the enabling condition

$$\Big((\chi(a) = \chi(b)) \Rightarrow \neg [\![b]\!]\Big) \wedge \Big((\chi(a) - 1 = \chi(b)) \Rightarrow \neg [\![a]\!]\Big) \tag{39}$$

Introducing $\delta \triangleq \chi(a) - \chi(b)$, condition 39 becomes

$$\Big((\delta = 0) \Rightarrow \neg [\![b]\!]\Big) \wedge \Big((\delta = 1) \Rightarrow \neg [\![a]\!]\Big) \tag{40}$$

The changes in state for the two precedence relations are the same, and therefore apply to the alternation relation as well.

$$\delta \leftarrow \begin{cases} \delta + 1 & \text{if } [\![a]\!] \wedge \neg [\![b]\!] \\ \delta - 1 & \text{if } \neg [\![a]\!] \wedge [\![b]\!] \\ \delta & \text{otherwise.} \end{cases} \tag{41}$$

Starting with the initial condition $\delta = 0$, from equations 40 and 41 we deduce that only two values $\delta = 0$ and $\delta = 1$ are possible. Therefore, alternation can be be specified by the FSM shown in figure 25, where S0 corresponds to the enabling condition $\delta = 0$, whereas state

S1 corresponds to $\delta = 1$. While the state machine associated with a precedence observer is infinite, the composition of the two precedence relations involved in an alternation results in a 2-state machine. This graphical specification of clock relations is exploited in Annex B.

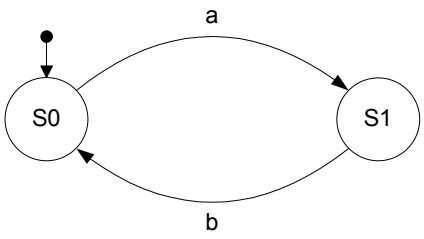


Figure 25: FSM of the strict alternation relation.

The violation condition for the alternation is the negation of the enabling condition 40:

$$\Big(\big(\delta = 0\big) \wedge [\![b]\!]\Big) \vee \Big(\big(\delta = 1\big) \wedge [\![a]\!]\Big) \tag{42}$$

The FSM of the alternation observer (figure 26) is obtained by adding a sink state (SV), and violation transitions, shown as red arcs. V is the violation signal. The original specification of the alternation observer (table 5) a priori needs integers for encoding δ. The FSM (figure 26) can be preferred for the observer implementation because it can be encoded with simple Boolean variables. Table 5 groups together the results in equational forms.

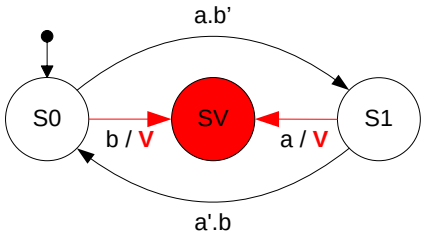


Figure 26: FSM detecting the violation of the strict alternation relation.

Synchronization $a \boxed{\bowtie} b$, which defined as $\big(a \boxed{\prec} b\$1\big) \mid \big(b \boxed{\prec} a\$1\big)$, can be transformed similarily. The details are skipped and gathered in table 5. There exist only three values for δ: 0, 1, and -1. States S0, SA, and SB correspond to conditions $\delta = 0$, $\delta = 1$, and $\delta = -1$ respectively. Here again, a FSM can be used to specify the behavior of the synchronization (figure 27). Note that in this FSM, when a and b tick simultaneously in state S0, the state is unchanged. This is why the transition has not been explicitly drawn.

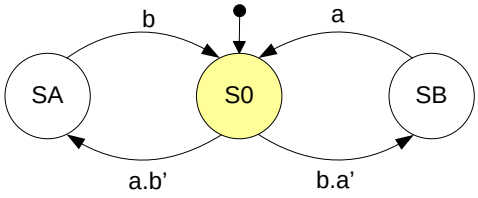


Figure 27: FSM of the strict synchronization relation.

Table 5: Violations of derived clock relations

Relation	Enabling	Changes	Violation
$a \boxed{=} b$	$[\![a]\!] = [\![b]\!]$		$[\![a]\!] \oplus [\![b]\!]$
$a \boxed{\sim} b$	$\big((\delta = 0) \Rightarrow \neg [\![b]\!]\big) \wedge \big((\delta = 1) \Rightarrow \neg [\![a]\!]\big)$	$\delta \leftarrow \begin{cases} 1 & \text{if } [\![a]\!] \wedge \neg [\![b]\!] \\ 0 & \text{if } \neg [\![a]\!] \wedge [\![b]\!] \\ \delta & \text{otherwise.} \end{cases}$	$\big((\delta = 0) \wedge [\![b]\!]\big) \vee \big((\delta = 1) \wedge [\![a]\!]\big)$
$a \boxed{\bowtie} b$	$\big((\delta = 1) \Rightarrow \neg [\![a]\!]\big) \wedge \big((\delta = -1) \Rightarrow \neg [\![b]\!]\big)$	$\delta \leftarrow \begin{cases} \delta + 1 & \text{if } [\![a]\!] \wedge \neg [\![b]\!] \\ \delta - 1 & \text{if } \neg [\![a]\!] \wedge [\![b]\!] \\ \delta & \text{otherwise.} \end{cases}$	$\big((\delta = 1) \wedge [\![a]\!]\big) \vee \big((\delta = -1) \wedge [\![b]\!]\big)$

where $\delta \triangleq \chi(a) - \chi(b)$

The observers for the various forms of alternation and synchronization are given in Annex B as FSM.

3.4 Generators for clock expressions

With each clock expression, we associate a *generator*. It generates clock ticks and maintains an internal state according to the presence or absence of ticks of its operand clocks. In this sub-section, generators are grouped according to the kind of their internal state. The results are presented in a tabular form: the expression in the first column, the condition to generate a tick in the second column, the possible changes in state in the third column, and the initial state in the last column. The last three columns reflect the operational semantics of CCSL [3].

3.4.1 Combinatorial generators

For these generators, emitting a tick results only from the presence/absence of incoming clock ticks. This group concerns the clock expressions *union*, *intersection*, and *difference*. No internal state is needed.

Table 6: Combinatorial generators

Expression	Ticks when	Changes	Initial
$a + b$	$[\![a]\!] \vee [\![b]\!]$		
$a * b$	$[\![a]\!] \wedge [\![b]\!]$		
$a - b$	$[\![a]\!] \wedge \neg [\![b]\!]$		

3.4.2 Clock index dependent generators

The clock expressions *sup* and *inf* depends on the difference (δ) between the indexes of the two clock operands. The integer δ is the internal state. It keeps track of the incoming ticks.

Table 7: Clock index dependent generators

Expression	Ticks when	Changes	Initial
$a \upharpoonright k$	$[\![a]\!]$	$\chi \leftarrow \begin{cases} \chi + 1 & \text{if } [\![a]\!] \\ \chi & \text{otherwise.} \end{cases}$	$\chi = 0$
$a \vee b$	$\big((\delta < 0) \wedge [\![a]\!]\big) \vee \big((\delta = 0) \wedge ([\![a]\!] \wedge [\![b]\!])\big) \vee \big((\delta > 0) \wedge [\![b]\!]\big)$	$\delta \leftarrow \begin{cases} \delta + 1 & \text{if } [\![a]\!] \wedge \neg [\![b]\!] \\ \delta - 1 & \text{if } \neg [\![a]\!] \wedge [\![b]\!] \\ \delta & \text{otherwise.} \end{cases}$	$\delta = 0$
$a \wedge b$	$\big((\delta < 0) \wedge [\![b]\!]\big) \vee \big((\delta = 0) \wedge ([\![a]\!] \vee [\![b]\!])\big) \vee \big((\delta > 0) \wedge [\![a]\!]\big)$	$\delta \leftarrow \begin{cases} \delta + 1 & \text{if } [\![a]\!] \wedge \neg [\![b]\!] \\ \delta - 1 & \text{if } \neg [\![a]\!] \wedge [\![b]\!] \\ \delta & \text{otherwise.} \end{cases}$	$\delta = 0$

where $\delta \triangleq \chi(a) - \chi(b)$

3.4.3 Generators for clocks with death

The clock expression $a \lightning b$ dies on the first tick of `b`. The internal state is a Boolean $isDead$. Boolean values are denoted as 0 and 1, for `false` and `true`, respectively.

Table 8: Generators for clocks with death

Expression	Ticks when	Changes	Initial
$a \lightning b$	$\neg isDead \wedge ([\![a]\!] \wedge \neg [\![b]\!])$	$isDead \leftarrow \begin{cases} 1 & \text{if } [\![b]\!] \\ isDead & \text{otherwise.} \end{cases}$	$isDead = 0$

3.4.4 Generators for clocks with schedule

Table 9: Generators for clocks with schedule

Expression	Ticks when	Changes	Initial
$a \hat{} n$	$[\![a]\!] \wedge (bw = 1)$	$bw \leftarrow \begin{cases} v & \text{if } [\![a]\!] \wedge (bw = 0.v) \\ 0 & \text{if } [\![a]\!] \wedge (bw = 1) \\ bw & \text{otherwise.} \end{cases}$	$bw = 0^{[\![n]\!]-1}.1$
$a \searrow b$	$[\![b]\!] \wedge (bw = 1)$	$bw \leftarrow \begin{cases} 1 & \text{if } [\![a]\!] \\ 0 & \text{if } [\![b]\!] \wedge \neg [\![a]\!] \\ bw & \text{otherwise.} \end{cases}$	$bw = 0$
$a \searrow b$	$[\![b]\!] \wedge ((bw = 1) \vee [\![a]\!])$	$bw \leftarrow \begin{cases} 1 & \text{if } [\![a]\!] \wedge \neg [\![b]\!] \\ 0 & \text{if } [\![b]\!] \\ bw & \text{otherwise.} \end{cases}$	$bw = 0$
$a \blacktriangledown w$	$[\![a]\!] \wedge (bw = 1.v)$	$bw \leftarrow \begin{cases} v & \text{if } [\![a]\!] \wedge (bw = b.v) \\ bw & \text{otherwise.} \end{cases}$	$bw = w$
$a \,(\, n \,) \rightsquigarrow b$	$[\![b]\!] \wedge (bw = 1.v)$	$bw \leftarrow \begin{cases} v & \text{if } [\![b]\!] \wedge \neg [\![a]\!] \wedge (bw = b.v) \\ v + 0^{[\![n]\!]-1}.1 & \text{if } [\![b]\!] \wedge [\![a]\!] \wedge (bw = b.v) \\ bw + 0^{[\![n]\!]-1}.1 & \text{if } [\![a]\!] \wedge \neg [\![b]\!] \\ bw & \text{otherwise.} \end{cases}$	$bw = 0$

where v is a binary word

3.4.5 Generators for clock concatenation

The concatenation $a \bullet b$ is rewritten into b when a dies.

Table 10: Generators for clock concatenation

Expression	Ticks when	Changes	Initial
$a \bullet expr2$	$[\![a]\!]$	$expr \leftarrow \begin{cases} expr2 & \text{if } a.isDead \\ expr & \text{otherwise.} \end{cases}$	$expr = a \bullet expr2$

3.5 Adaptors

An adaptor has to generate ticks of a c_clock according to the state of one or several objects of the program. Collectively, the states of these objects characterize an event of interest for the system. Such events are greatly dependent on the application, and the way to detect their occurrences is language specific. So, there is little to say about adaptors in general.

4 Esterel implementation

In this section we propose a library of Esterel modules that implement the general concepts of adaptors, observers, and generators introduced in the previous chapter. The code is written in Esterel v7, version v7_60 for Esterel Studio 6.1.

4.1 Data and interface units

Esterel allows separate programming of data, interface, and module units. Genericity is also supported. Existence of various units and genericity are exploited in our implementation of CCSL observers in Esterel.

C_clocks

To implement CCSL observers in Esterel, we have first to decide how to represent a CCSL clock. We have chosen a *pair of pure signals*. The presence of the signal 'presence' represents a tick of the associated clock. The second signal 'alive' reflects the status of the clock: the clock is alive when the signal is present, whereas the clock is not existing (not yet created or dead) when the signal is absent. The two signals are grouped together in a *port*, and a c_clock is a port typed by `C_Clock_Intf`.

```
interface C_Clock_Intf:
  output presence;
  output alive;
end interface
```

Other data and interface units will be defined in the following sub-sections.

4.2 Adaptors

In Esterel a logical clock can be associated with any signal. This clock ticks whenever the signal is present. Hence, an Esterel adaptor maps an Esterel signal to a c_clock. We have to distinguish between pure and valued signals.

4.2.1 Pure signal adaptor

For a pure signal `A`, the adaptor code is obvious:

```
1  module Ccsl_A_pure:
2      input A;
3      port c_A: C_Clock_Intf;
4
5      sustain {
6          c_A.alive,
7          c_A.presence if A
8      }
```

```
9
10  end module
```

Note that `c_A.alive` is emitted at each instant (line 6) when this module is active.

4.2.2 Valued signal adaptor

For a valued signal, the adaptor code is also very simple. It is a generic module which considers only the presence status of the input signal.

```
1  module Ccsl_A_valued:
2      generic type T;
3      input A: T;
4      port c_A: C_Clock_Intf;
5
6      sustain {
7          c_A.alive,
8          c_A.presence if A
9      }
10
11 end module
```

In line 2, a generic type is introduced. It types signal A (line 3). The actual type must be given at the module instantiation. When the module is active, signal c_A.presence is emitted whenever the input signal A is present (line 8).

4.2.3 Rising-edge signal adaptor

Since the presence status of a signal is not persistent, hand-shake in Esterel is often implemented as

```
abort
  sustain resquest
when response
```

The actual event in this case is the "rising edge" of signal `request` (*i.e.,* a signal present at the first instant of the sustain). The adaptor Ccsl_A_risingEdge represents this behavior (Synchart in figure 28).

Note that the first two adaptors are so simple that their code is usually in-lined.

4.3 Observers

4.3.1 Observer interface

Each CCSL relation observer has two input c_clocks and one output that signals possible violations. This fix interface can be specified by an interface.

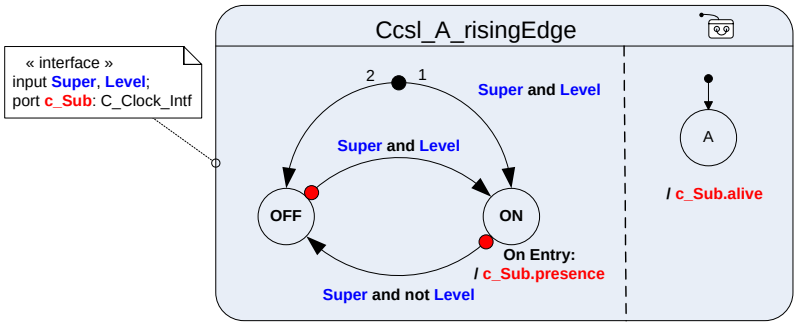


Figure 28: SyncChart of the Rising Edge adaptor.

```
1  interface Ccsl_R_Intf:
2    port c_A: mirror C_Clock_Intf;
3    port c_B: mirror C_Clock_Intf;
4    output Violation;
5  end interface
```

Port `c_A` is typed "**mirror** C_Clock_Intf" (line 2). This means that the port consists of the signals declared in interface `C_Clock_Intf` but with their direction reverted (imposed by the keyword '**mirror**'). It is the same for port `c_b`, line 3. As for Violation, it is a pure signal emitted as soon as a violation occurs. Each *Observer module* includes this interface in its code (statement "**extends** Ccsl_R_Intf;").

4.3.2 Subclocking

```
1  module Ccsl_R_subclock:
2  extends Ccsl_R_Intf;
3    sustain Violation if c_A.presence and not c_B.presence
4  end module
```

The statement **sustain** (line 4) emits `Violation` if `c_A` ticks while `c_B` does not. This condition is directly extracted from table 4, first row, last column.

4.3.3 Tight subclocking

This clock relation should be tested when `c_a` is alive. In Esterel, this is easily done by waiting for `c_a` to be born (line 3), and checking the violation (line 5) while `c_a` is alive (lines 4–6).

```
1  module Ccsl_R_tightsubclock:
2  extends Ccsl_R_Intf;
3  await immediate c_A.alive;
4  abort
5    sustain Violation if c_A.presence xor c_B.presence
```

```
6  when not c_A.alive
7  end module
```

Line 5 encodes $\mathsf{V} = (\mathsf{c\_a} \oplus \wedge \mathsf{c\_b})$ from table 4.

4.3.4 Exclusion

The third clock relation observer emits Violation if both `c_A` and `c_B` tick.

```
module Ccsl_R_exclusive:
extends Ccsl_R_Intf;
  sustain Violation if c_A.presence and c_B.presence
end module
```

4.3.5 Precedences

The precedence observers are more complex. They have to maintain an internal counter (delta). The strict and non strict version have common behavior and they differ on their violation conditions (see table 4). Their code can be combined and the discremination done by a generic constant set at the observer module instantiation. The discreminant is the the generic constant `KIND`, the type of which is a the enumeration `Ccsl_Strictness_Kind` defined in a data unit.

```
data Ccsl_Types:
type Ccsl_Strictness_Kind =
  enum {
    STRICT=0, NONSTRICT=1, LNS=2, RNS=3
  };
end data
```

The precedence observer is a generic module.

```
1  module Ccsl_R_precedes:
2  extends Ccsl_Types;
3  extends Ccsl_R_Intf;
4
5  constant KIND: Ccsl_Strictness_Kind;
6  generic constant UMAX: unsigned;
7  type CounterType = unsigned<UMAX>;
8
9  signal Delta: value CounterType init 0 in
10   sustain {
11     Violation if
12       (pre(?Delta) = 0) and c_B.presence and
13       ((KIND = STRICT) or ((KIND=NONSTRICT) and not c_A.presence)),
14     ?Delta <= sat<UMAX>(pre(?Delta) + 1) if
15       not Violation and c_A.presence and not c_B.presence,
16     ?Delta <= pre(?Delta) - 1 if
```

```
17          not Violation and not c_A.presence and c_B.presence
18      }
19    end signal
20
21    end module
```

This observer module deserves explanations:

- two constants must be provided at the module instantiation: `KIND` and `UMAX`; the latter is the expected maximal value for `Delta`;
- `CounterType` is a generic type which contains natural numbers from 0 to `UMAX-1`;
- `Delta` counts the difference on the numbers of ticks of `c_a` and `c_b` (lines 15–18);
- at each instant the violation condition is tested (lines 12–14);
- the value of `Delta` is changed only if there is no violation, this ensures that `Delta` is never decremented below 0 (line 17).
- since Esterel applies strict arithmetic rules, we have to assert that `Delta` never exceeds value `UMAX-1`, hence the use of the **sat**<UMAX>() predefined function.

4.3.6 Equality

This clock relation is similar to the tight sub-clocking on page 30, but must be *always* valid: there is no longer conditions on the birth and death of clocks.

```
1  module Ccsl_R_equal:
2  extends Ccsl_R_Intf;
3    sustain Violation if c_A.presence xor c_B.presence
4  end module
```

4.3.7 Alternation

The violation conditions for the four variants of alternation are given in annex B.1. They are specified as FSMs. In Esterel they are better specified as SyncCharts. The usage of explicit priority on transition makes the arc labels simpler. For instance, figure 29) is the syncChart equivalent to the FSM in figure 33. Both check the strict alternation.

A generic module specifies the four variants.

```
1  module Ccsl_R_alternates:
2  extends Ccsl_Types;
3  extends Ccsl_R_Intf;
4  constant KIND: Ccsl_Strictness_Kind;
5
6  signal Turn: value bool init '0 in
7    loop
```

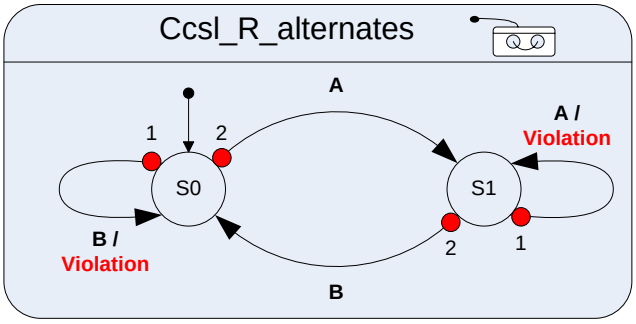


Figure 29: SyncChart of the alternation observer.

```
8       if pre(?Turn) then
9         //state S1
10        if (KIND = STRICT) or (KIND = RNS) then
11          if
12          case c_A.presence do
13            emit Violation
14          case c_B.presence do
15            emit Turn('0)
16          end if
17        else // (KIND = LNS) or (KIND = NONSTRICT)
18          if
19          case c_A.presence and not c_B.presence do
20            emit Violation
21          case c_A.presence and c_B.presence
22          case c_B.presence do
23            emit Turn('0)
24          end if
25        end if
26      else
27        //state S0
28        if (KIND = STRICT) or (KIND = LNS) then
29          if
30          case c_B.presence do
31            emit Violation
32          case c_A.presence do
33            emit Turn('1)
34          end if
35        else // (KIND = RNS) or (KIND = NONSTRICT)
36          if
37          case not c_A.presence and c_B.presence do
38            emit Violation
```

```
39          case c_A.presence and c_B.presence
40          case c_A.presence do
41            emit Turn('1)
42          end if
43        end if
44      end if //Turn
45    each tick
46
47  end signal
48
49  end module
```

This is a hand-encoded version of the syncCharts, it is easier to read than the automated translation. Of course, such a textual version of SyncCharts is not required, the Esterel compiler directly deals with the graphical specification. The above textual version is just to show how a purely textual specification is possible. The two states S0 and S1 have been encoded as a **bool** signal 'Turn' initialized to '0 (line 6). The conditions are evaluated at each instant (statement **loop** ... **each tick**, in lines 7–45). For each state, the outgoing conditions are written in a decreasing order of priority (statement **if case** ... **do end if**). The first matching transition is taken. A **case** ... without a **do** ... means that nothing has to be executed so that no violation is emitted and the state is left unchanged. The effective changes in state are caused by emitting the next state value (*i.e.*, **emit** Turn('1) on line 41).

4.3.8 Synchronization

The violation conditions for the four variants of synchronization are given as FSMs in annex B.2. The following Esterel module encodes the four FSMs.

```
1  module Ccsl_R_synchronizes:
2  extends Ccsl_Types;
3  extends Ccsl_R_Intf;
4  constant KIND: Ccsl_Strictness_Kind;
5  type State = enum {S0=0, SA=1, SB=2};
6
7  signal S:State init S0 in
8    loop
9      if
10     case pre(?S)=S0 do // in S0
11       if
12       case c_A.presence and not c_B.presence do
13         emit S(SA)
14       case c_B.presence and not c_A.presence do
15         emit S(SB)
16       end if
17     case pre(?S)=SA do // in SA
18       if (KIND = STRICT) or (KIND = RNS) then
```

```
19          if
20          case c_A.presence do
21            emit Violation
22          case c_B.presence do
23            emit S(S0)
24          end if
25        else // (KIND = NONSTRICT) or (KIND = LNS)
26          if
27          case c_A.presence and not c_B.presence do
28            emit Violation
29          case c_B.presence and not c_A.presence do
30            emit S(S0)
31          end if
32        end if
33      case pre(?S)=SB do  // in SB
34        if (KIND = STRICT) or (KIND = LNS) then
35          if
36          case c_B.presence do
37            emit Violation
38          case c_A.presence do
39            emit S(S0)
40          end if
41        else  // (KIND = NONSTRICT) or (KIND = RNS)
42          if
43          case c_B.presence and not c_A.presence do
44            emit Violation
45          case c_A.presence and not c_B.presence do
46            emit S(S0)
47          end if
48        end if
49      end if
50    each tick
51  end signal
52  end module
```

4.4 Generators

4.4.1 CCSL expressions

4.4.2 Union

```
1  module Ccsl_E_union:
2  port c_A: mirror C_Clock_Intf;
3  port c_B: mirror C_Clock_Intf;
4  port c_O: C_Clock_Intf;
5    abort
```

```
6       sustain {
7         c_O.alive,
8         c_O.presence if
9         c_A.presence or c_B.presence
10      }
11    when not (c_A.alive or c_B.alive)
12  end module
```

The union expression dies when both input clocks die.

4.4.3 Intersection

```
1   module Ccsl_E_inter:
2   port c_A: mirror C_Clock_Intf;
3   port c_B: mirror C_Clock_Intf;
4   port c_O: C_Clock_Intf;
5     abort
6       sustain {
7         c_O.alive,
8         c_O.presence if
9         c_A.presence and c_B.presence
10      }
11    when not (c_A.alive and c_B.alive)
12  end module
```

The inter expression dies when any input clock dies.

4.4.4 Difference

```
1   module Ccsl_E_minus:
2   port c_A: mirror C_Clock_Intf;
3   port c_B: mirror C_Clock_Intf;
4   port c_O: C_Clock_Intf;
5     abort
6       sustain {
7         c_O.alive,
8         c_O.presence if
9         c_A.presence and not c_B.presence
10      }
11    when not c_A.alive
12  end module
```

The minus expression dies when the first input clock dies.

4.4.5 As from

```
1   module Ccsl_E_asfrom:
2   generic constant UMAX:unsigned;
```

```
3  generic constant K:unsigned;
4  type Unsigned_t = unsigned<UMAX>;
5  port c_A: mirror C_Clock_Intf;
6  port c_O: C_Clock_Intf;
7    signal Chi:value Unsigned_t init 0 in
8      abort
9        if K = 0 else
10         weak abort
11           sustain
12             ?Chi <= assert<UMAX-1>(pre(?Chi))+1 if c_A.presence
13         when ?Chi = K
14       end if;
15       sustain {
16         c_O.alive,
17         c_O.presence if c_A.presence,
18         ?Chi <= assert<UMAX-1>(pre(?Chi))+1 if c_A.presence
19       }
20     when immediate not c_A.alive
21   end signal
22 end module
```

The asfrom expression dies when the input clock dies.

4.4.6 Sup

```
1  module Ccsl_E_sup:
2  generic constant SMAX:unsigned;
3  type Signed_t = unsigned<SMAX>;
4  port c_A: mirror C_Clock_Intf;
5  port c_B: mirror C_Clock_Intf;
6  port c_O: C_Clock_Intf;
7  signal Delta:value Signed_t init 0 in
8    abort
9      sustain {
10       // changes in internal state
11       ?Delta <= sat<SMAX>(pre(?Delta)+1) if
12         c_A.presence and not c_B.presence,
13       ?Delta <= sat<SMAX>(pre(?Delta)-1) if
14         c_B.presence and not c_A.presence,
15       // ticks
16       c_O.alive,
17       c_O.presence if (pre(?Delta) < 0) and c_A.presence,
18       c_O.presence if (pre(?Delta) > 0) and c_B.presence,
19       c_O.presence if (pre(?Delta) = 0) and c_A.presence and c_B.presence
20     }
21   when immediate
22     ((pre(?Delta < 0) and not c_A.alive) or
```

```
23      ((pre(?Delta > 0) and not c_B.alive)
24  end signal
25  end module
```

The sup expression dies when the clock with the lower index dies, *i.e.,* a if $\delta < 0$, or b is $\delta > 0$.

4.4.7 Inf

```
1   module Ccsl_E_inf:
2   generic constant SMAX:unsigned;
3   type Signed_t = unsigned<SMAX>;
4   port c_A: mirror C_Clock_Intf;
5   port c_B: mirror C_Clock_Intf;
6   port c_O: C_Clock_Intf;
7   signal Delta:value Signed_t init 0 in
8     abort
9       sustain {
10        // changes in internal state
11        ?Delta <= sat<SMAX>(pre(?Delta)+1) if
12          c_A.presence and not c_B.presence,
13        ?Delta <= sat<SMAX>(pre(?Delta)-1) if
14          c_B.presence and not c_A.presence,
15        // ticks
16        c_O.alive,
17        c_O.presence if (pre(?Delta) > 0) and c_A.presence,
18        c_O.presence if (pre(?Delta) < 0) and c_B.presence,
19        c_O.presence if (pre(?Delta) = 0) and (c_A.presence or c_B.presence)
20      }
21    when immediate not (c_A.alive or c_B.alive)
22  end signal
23  end module
```

The inf expression dies when both input clocks die.

4.4.8 Upto

```
1   module Ccsl_E_upto:
2   port c_A: mirror C_Clock_Intf;
3   port c_B: mirror C_Clock_Intf;
4   port c_O: C_Clock_Intf;
5   if c_A.alive then
6     abort
7       sustain {
8         c_O.alive,
9         c_O.presence if c_A.presence
10      }
11    when c_B.presence or not c_A.alive
```

```
12  end if
13  end module
```

The upto expression dies when b ticks or a dies.

4.4.9 Await

```
1   module Ccsl_E_await:
2   generic constant N: unsigned;
3   port c_A: mirror C_Clock_Intf;
4   port c_B: mirror C_Clock_Intf;
5   port c_O: C_Clock_Intf;
6   if c_A.alive then
7     abort
8       abort
9         sustain c_O.alive
10      when N c_A.presence;
11      emit c_O.presence
12      // then die
13    when not c_A.alive
14  end if
15  end module
```

The await expression dies after n ticks of a or when a dies.

4.4.10 Sample

```
1   module Ccsl_E_sample:
2   extends Ccsl_Types;
3
4   constant KIND: Ccsl_Strictness_Kind;
5   port c_A: mirror C_Clock_Intf;
6   port c_B: mirror C_Clock_Intf;
7   port c_O: C_Clock_Intf;
8   if c_A.alive and c_B.alive then
9     signal Bw: value bool init '0 in
10      abort
11        sustain {
12          c_O.alive,
13          // internal state management
14          ?Bw <= '1 if c_A.presence and
15            (
16              (KIND = STRICT) or
17              ((KIND = NONSTRICT) and not c_B.presence)
18            ),
19          ?Bw <= '0 if c_B.presence and
20            (
21              ((KIND = STRICT) and not  c_A.presence)
```

```
22              or (KIND = NONSTRICT)
23            ),
24          // tick
25          c_O.presence if c_B.presence and
26            (
27              pre(?Bw) or
28              ((KIND = NONSTRICT) and c_A.presence)
29            )
30        }
31
32      when not (c_B.alive and (pre(?Bw) or c_A.alive))
33    end signal
34  end if
35  end module
```

The sample expression dies when b dies or when a is dead and the schedule (Bw) is empty.

4.4.11 Defer

```
1   module Ccsl_E_defer:
2   generic constant DELAYMAX: unsigned;
3   type T = unsigned<DELAYMAX>;
4   type Bv_t = bool[DELAYMAX];
5
6   port c_A: mirror C_Clock_Intf;
7   port c_B: mirror C_Clock_Intf;
8   port c_O: C_Clock_Intf;
9   input N: T;
10  constant Bw0: Bv_t = resize('b0,DELAYMAX);
11
12  if c_A.alive and c_B.alive then
13    signal Bw: value Bv_t init '0, Update: value Bv_t in
14      abort
15        sustain {
16          c_O.alive,
17          // internal state management
18          ?Update <= u2onehot(?N-1,DELAYMAX) if c_A.presence,
19          ?Bw <= (pre(?Bw) >> 1) if c_B.presence and
20            not c_A.presence,
21          ?Bw <= ((pre(?Bw) >> 1) [or] ?Update) if
22            c_A.presence and c_B.presence,
23          ?Bw <= (pre(?Bw) [or] ?Update) if
24            c_A.presence and not c_B.presence,
25          // tick
26          c_O.presence if c_B.presence and pre(?Bw[0])
27        }
28      when not (c_B.alive and ((pre(?Bw) = Bw0) or c_A.alive))
```

```
29    end signal
30  end if
31  end module
```

4.4.12 Filter

```
1   interface Ccsl_E_filteredBy_Intf:
2     generic constant SIZE: unsigned;
3     type Bv_t = bool[SIZE];
4     input Bw: value Bv_t;
5     port c_Super: mirror C_Clock_Intf;
6     port c_Sub: C_Clock_Intf;
7   end interface
```

```
1   module Ccsl_E_Pref_filter:
2   generic constant PREF_SIZE: unsigned;
3   extends Ccsl_E_filteredBy_Intf [
4     constant PREF_SIZE/SIZE;
5     type PREF_Bv_t/Bv_t];
6
7   if c_Super.alive then
8     abort
9       var SR: PREF_Bv_t := ?Bw in
10        trap T in
11          sustain c_Sub.alive
12        ||
13          repeat PREF_SIZE times
14            await c_Super.presence;
15            emit c_Sub.presence if SR[0];
16            SR := SR >> 1
17          end repeat;
18          exit T
19        end trap
20      end var
21    when not c_Super.alive
22  end if
23
24  end module
```

```
1   module Ccsl_E_Per_filter:
2   generic constant PER_SIZE: unsigned;
3   extends Ccsl_E_filteredBy_Intf [
4     constant PER_SIZE/SIZE;
5     type PER_Bv_t/Bv_t];
6
7   if c_Super.alive then
8     abort
```

```
9       var SR: PER_Bv_t := ?Bw, Carry:bool in
10        sustain c_Sub.alive
11      ||
12        loop
13          await c_Super.presence;
14          Carry := SR[0];
15          emit c_Sub.presence if Carry;
16          SR := SR >> 1;
17          SR[PER_SIZE-1] := Carry
18        end loop
19      end var
20    when not c_Super.alive
21  end if
22
23  end module
```

```
1   module Ccsl_E_filter:
2   generic constant PREF_SIZE: unsigned;
3   generic constant PER_SIZE: unsigned;
4   extends data Ccsl_E_filteredBy_Intf [
5     constant PREF_SIZE/SIZE;
6     type Pref_Bv_t/Bv_t];
7   extends data Ccsl_E_filteredBy_Intf [
8     constant PER_SIZE/SIZE;
9     type Per_Bv_t/Bv_t];
10
11
12  input Pref: value Pref_Bv_t;
13  input Per: value Per_Bv_t;
14
15  port c_Super: mirror C_Clock_Intf;
16  port c_Sub: C_Clock_Intf;
17
18  abort
19    // prefix
20    run Ccsl_E_Pref_filter [signal Pref/Bw];
21    // period
22    run Ccsl_E_Per_filter [signal Per/Bw]
23  when not c_Super.alive
24
25  end module
```

The filter expression dies when the super clock dies or if the period is empty when the prefix terminates.

4.5 Extended observers

This sub-section shows how the library of Esterel modules can be extended. We have chosen to illustrate this capability with the *by-packet*-relations. These relations are useful in Digital Signal Processing applications (*e.g.*, see the signal filtering described in [5]). The principle is to consider "packets" of ticks, instead of individual ticks. The size of the packets is a constant.

Figure 30 represents the strict by-packet alternation where the ticks of a are grouped together by $\alpha = 4$, and the ticks of b by $\beta = 3$. This is symbolically written $a/\alpha \boxed{\sim} b/\beta$ and read "a by α strictly alternates with b by β.

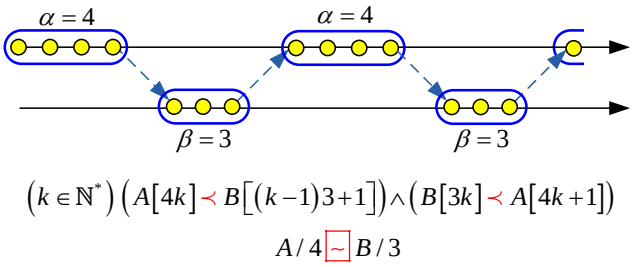


Figure 30: Example of strict by-packet alternation relation.

To deal with the by-packet relations, we introduce two auxiliary clock expressions: firstBy and lastBy. The associated firstBy generator ticks at the first instant of each packet, whereas the lastBy generator ticks at the last instant of each packet. Equations 43 and 44 specify these clock expressions.

$$(\forall k \in \mathbb{N}^\star)\big(a.\ \mathsf{firstBy}\ (\alpha)[k] \equiv a[(k-1)\alpha + 1]\big) \tag{43}$$

$$(\forall k \in \mathbb{N}^\star)\big(a.\ \mathsf{lastBy}\ (\alpha)[k] \equiv a[k\alpha]\big) \tag{44}$$

They can be seen as special cases of filtering (Eqs 45 and 46):

$$\big(a.\ \mathsf{firstBy}\ (\alpha)\big) \boxed{=} \big(a \blacktriangledown \big(1.0^{\alpha-1}\big)^{\omega}\big) \tag{45}$$

$$\big(a.\ \mathsf{lastBy}\ (\alpha)\big) \boxed{=} \big(a \blacktriangledown \big(0^{\alpha-1}.1\big)^{\omega}\big) \tag{46}$$

The Esterel modules (sub-section 4.5.1) implement these generators.

4.5.1 By-packet generators

```
1 module Ccsl_E_firstBy:
2 port c_Super: mirror C_Clock_Intf;
3 port c_Sub: C_Clock_Intf;
```

```
4  constant SIZE: unsigned;
5
6  if c_Super.alive then
7      //assert posSIZE = (SIZE > 0);
8      abort
9          sustain c_Sub.alive
10     ||
11         if static SIZE = 1 then
12             sustain c_Sub.presence if c_Super.presence
13         else
14             await immediate c_Super.presence;
15             emit c_Sub.presence;
16             every SIZE c_Super.presence do
17                 emit c_Sub.presence
18             end every
19         end if
20     when not c_Super.alive
21 end if
22
23 end module
```

```
1  module Ccsl_E_lastBy:
2  port c_Super: mirror C_Clock_Intf;
3  port c_Sub: C_Clock_Intf;
4  constant SIZE: unsigned;
5
6  if c_Super.alive then
7      //assert posSIZE = (SIZE > 0);
8      abort
9          sustain c_Sub.alive
10     ||
11         if static SIZE = 1 then
12             sustain c_Sub.presence if c_Super.presence
13         else
14             await immediate c_Super.presence;
15             await (SIZE-1) c_Super.presence;
16             loop
17                 emit c_Sub.presence
18             each SIZE c_Super.presence
19         end if
20     when not c_Super.alive
21 end if
22
23 end module
```

4.5.2 By-packet alternation

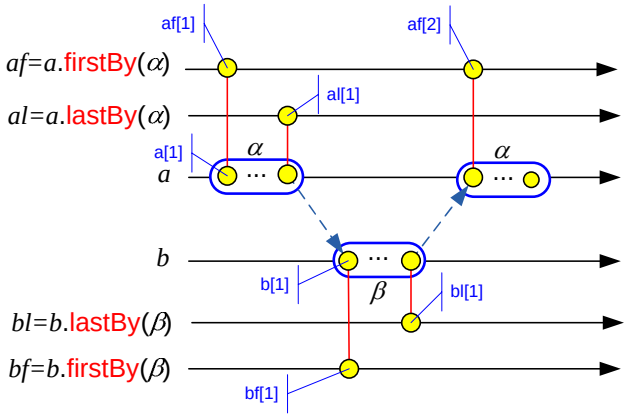


Figure 31: Strict by-packet alternation instant ordering.

Consider relation $a/\alpha \boxed{\sim} b/\beta$. Let $af \triangleq a.\ \mathsf{firstBy}\ (\alpha)$ and $al \triangleq a.\ \mathsf{lastBy}\ (\alpha)$. bf and bl are defined in a similar way. Referring to figure 31, the following relations hold:

$$af \boxed{\preccurlyeq} al \tag{47}$$

$$al \boxed{\prec} bf \tag{48}$$

$$bf \boxed{\preccurlyeq} bl \tag{49}$$

$$bl \boxed{\prec} af\$1 \tag{50}$$

Relations 47 and 49 are trivially true because of the total ordering over the instants of a clock. We used the non strict precedence instead of the strict one to cover the case of a packet of size 1 (in which case the first and last instants of a packet are coincident). From these relations we deduce that al strictly alternates with bf (equation 51) and af strictly alternates with bl (equation 52).

$$\dfrac{\dfrac{(bf \boxed{\preccurlyeq}^{(49)} bl)\quad (bl \boxed{\prec}^{(50)} af\$1)}{bf \boxed{\prec} al\$1}\quad \dfrac{af \boxed{\preccurlyeq}^{(47)} al}{af\$1 \boxed{\preccurlyeq} al\$1}\quad (al \boxed{\prec}^{(48)} bf)}{al \boxed{\sim} bf} \tag{51}$$

$$\dfrac{\dfrac{(af \overset{(47)}{\boxed{\preccurlyeq}} al)\quad (al \overset{(48)}{\boxed{\prec}} bf)\quad (bf \overset{(49)}{\boxed{\preccurlyeq}} bl)}{af \boxed{\prec} bl}\qquad (bl \overset{(50)}{\boxed{\prec}} af\$1)}{af \boxed{\sim} bl} \tag{52}$$

Conversely, assuming $af \boxed{\sim_=} al$, $bf \boxed{\sim_=} bl$, $al \boxed{\sim} bf$, and $af \boxed{\sim} bl$, we get equations 47–50. So, $a \,/\, \alpha \boxed{\sim} b \,/\, \beta$ is equivalent to the conjunction of the four relations: $af \boxed{\sim_=} al$, $al \boxed{\sim} bf$, $bf \boxed{\sim_=} bl$, and $af \boxed{\sim} bl$. The implementation of the corresponding observer is made by instantiating the previously given Esterel module (Ccsl_R_alternates, on page 32). For the sake of efficiency, the four modules can be combined into one that implements the synchronous product of the associated FSM. Figure 41 on page 56 in Annex B.3, is the resulting FSM.

Like alternation, the by-packet alternation has four variants:

$$a \,/\, \alpha \boxed{\sim} b \,/\, \beta \Leftrightarrow (af \boxed{\sim_=} al) \mid (al \boxed{\sim} bf) \mid (bf \boxed{\sim_=} bl) \mid (af \boxed{\sim} bl) \tag{53}$$

$$a \,/\, \alpha \boxed{\sim_=} b \,/\, \beta \Leftrightarrow (af \boxed{\sim_=} al) \mid (al \boxed{\sim_=} bf) \mid (bf \boxed{\sim_=} bl) \mid (af \boxed{\sim} bl) \tag{54}$$

$$a \,/\, \alpha \boxed{=\!\sim} b \,/\, \beta \Leftrightarrow (af \boxed{\sim_=} al) \mid (al \boxed{\sim} bf) \mid (bf \boxed{\sim_=} bl) \mid (af \boxed{=\!\sim} bl) \tag{55}$$

$$a \,/\, \alpha \boxed{=\!\sim\!=} b \,/\, \beta \Leftrightarrow (af \boxed{\sim_=} al) \mid (al \boxed{=\!\sim\!=} bf) \mid (bf \boxed{\sim_=} bl) \mid (af \boxed{=\!\sim\!=} bl) \tag{56}$$

The corresponding FSMs are provided in Annex B.3.

4.5.3 By-packet synchronization

The extension of synchronization to *by-packet* synchronization follows the same approach as for alternation. We just give an illustration of strict by-packet synchronization (figure 32).

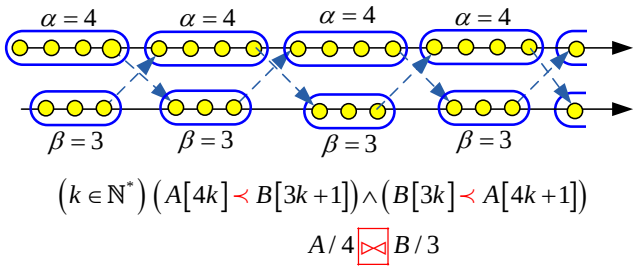


Figure 32: Example of strict by-packet synchronization relation.

Consider relation $a/\alpha \boxed{\bowtie} b/\beta$. Let $af \triangleq a.\,\mathsf{firstBy}\ (\alpha)$ and $al \triangleq a.\,\mathsf{lastBy}\ (\alpha)$. bf and bl are defined in a similar way. The following relations hold:

$$af \boxed{\preccurlyeq} al \tag{57}$$
$$al \boxed{\prec} bf\$1 \tag{58}$$
$$bf \boxed{\preccurlyeq} bl \tag{59}$$
$$bl \boxed{\prec} af\$1 \tag{60}$$

Remind that $a \boxed{\bowtie} b = \left(a \boxed{\prec} b\$1\right) \mid \left(b \boxed{\prec} a\$1\right)$. From equations 57 to 60 we deduce $af \boxed{\bowtie} bf$ (Eq 61) and $al \boxed{\bowtie} bl$ (Eq 62).

$$\frac{\dfrac{\overset{(57)}{\left(af \boxed{\preccurlyeq} al\right)} \quad \overset{(58)}{\left(al \boxed{\prec} bf\$1\right)}}{af \boxed{\prec} bf\$1} \quad \dfrac{\overset{(59)}{\left(bf \boxed{\preccurlyeq} bl\right)} \quad \overset{(60)}{\left(bl \boxed{\prec} af\$1\right)}}{bf \boxed{\prec} af\$1}}{af \boxed{\bowtie} bf} \tag{61}$$

$$\frac{\dfrac{\left(\overset{(58)}{al \boxed{\prec} bf\$1}\right) \quad \dfrac{\overset{(59)}{bf \boxed{\preccurlyeq} bl}}{bf\$1 \boxed{\preccurlyeq} bl\$1}}{al \boxed{\prec} bl\$1} \quad \dfrac{\left(\overset{(60)}{bl \boxed{\prec} af\$1}\right) \quad \dfrac{\overset{(57)}{af \boxed{\preccurlyeq} al}}{af\$1 \boxed{\preccurlyeq} al\$1}}{bl \boxed{\prec} al\$1}}{al \boxed{\bowtie} bl} \tag{62}$$

Hence, $a \ / \ \alpha \boxed{\bowtie} b \ / \ \beta$ is the conjunction of four relations: $af \boxed{\sim_=} al$, $af \boxed{\bowtie} bf$, $bf \boxed{\sim_=} bl$, and $al \boxed{\bowtie} bl$. The implementation of the corresponding observer is made by instantiating the Esterel modules Ccsl_R_alternates and Ccsl_R_synchronizes. The three other variants of the by-packet synchronization are left as exercises for the reader.

5 Future work

The continuation of this work is twofold—to consolidate the semantics of CCSL, and to apply the CCSL observer techniques to other languages.

Semantics

In section 2 we have given mathematical characterizations of CCSL relations and expressions. On the other hand, in section 3, we started with the structural operation semantics of CCSL, which has been introduced in a previous report [3], to determine the violation conditions of CCSL constraints. The formal link between the first (denotational) semantics and the second (operational) semantics of CCSL is still to be established.

The concepts of birth and death of a clock have also to be formally introduced in both semantics. The operational semantics has a form of death through its rewriting rules (when

an expression is rewritten as **0**). However the propagation of death through constraints, which is effectively used in several modules of section 4, has not been formally specified yet.

A third necessary improvement in CCSL is the introduction of local clock constraints. The relation tight subclocking on page 7 and the expression As from on page 13 were not defined in the original operational semantics. They have been introduced to open the way to local clock constraints.

Libraries for observers

Section 3 has explained how to build CCSL *observers*, *generators*, and *adaptors* for any CCSL constraints. A comprehensive library of Esterel modules has been provided in section 4. Esterel relies on the *perfect synchrony* hypothesis. As a consequence, this language has a well-defined notion of instant, and at each reaction, any signal has a unique status. This is not the case with non-strictly synchronous languages. Languages with *micro-step* semantics, like VHDL or systemC, have a notion of (simulation) instant, but an instant may consist of several "δ-cycles". The status of a signal may differ at two δ-cycles of the same simulation instant. This may lead to false-violations. So, δ-delay insensitive observers and generators must be implemented. Such a library for CCSL observers/generators in VHDL will be released soon.

References

[1] OMG. *UML Profile for MARTE, v1.0.* Object Management Group, November 2009. Document number: formal/2009-11-02.

[2] C. André, F. Mallet, and R. de Simone. Modeling time(s). In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 559–573. Springer, 2007.

[3] Charles André. Syntax and semantics of the clock constraint specification language (CCSL). Research Report 6925, INRIA, 05 2009.

[4] Frédéric Mallet, Marie-Agnès Peraldi-Frati, and Charles André. Marte CCSL to execute East-ADL timing requirements. In *ISORC* [18], pages 249–253.

[5] Charles André and Frédéric Mallet. Specification and verification of time requirements with CCSL and esterel. In *LCTES*, pages 167–176. ACM, June 2009.

[6] F. Boussinot and R. De Simone. The ESTEREL language. *Proceeding of the IEEE*, 79(9):1293–1304, September 1991.

[7] Gérard Berry. *The Esterel Language Primer, version v5_ 91.* Ecole des Mines de Paris, CMA, INRIA, July 2000.

[8] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, Amsterdam, 1993.

[9] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[10] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *AMAST '93: Proceedings of the Third International Conference on Methodology and Software Technology*, pages 83–96, London, UK, 1994. Springer-Verlag.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[12] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceeding of the IEEE*, 79(9):1270–1282, September 1991.

[13] C. André, F. Mallet, and M.-A. Peraldi-Frati. A multiform time approach to real-time system modeling: Application to an automotive system. In Universidade Nova de Lisboa, editor, *Int. Symp. on Industrial Embedded Systems*, pages 234–241, Lisboa, Portugal, July 2007. IEEE.

[14] Frédéric Mallet and Charles André. On the semantics of UML/marte clock constraints. In *ISORC* [18], pages 305–312.

[15] Charles André and Frédéric Mallet. Clock constraint specification language in UML/-MARTE CCSL. Research Report 6540, INRIA, 05 2008.

[16] Charles André and Frédéric Mallet. Modèles de contraintes temporelles pour systèmes polychrones. *JESA*, 43(7–8–9):725–739, 2009.

[17] W. Reisig. *Petri nets: an introduction.* Monograph on Theoretical Computer Science. Springer-Verlag, Berlin, 1985.

[18] IEEE. *12th IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC 2009).* IEEE Computer Society, March 2009.

A Binary Words

A.1 Finite/infinite binary words

Definition A.1 (Set of bit values)**.** $\mathbb{B} = \{0, 1\}$.

Definition A.2 (Finite binary word)**.** A *finite binary word* is a word of $(0+1)^*$.

Definition A.3 (Infinite binary word)**.** An *infinite binary word* is a word of $(0+1)^\omega$.

Definition A.4 (Periodic binary word)**.** A *periodic binary word* is an infinite binary word defined by the following grammar:

$$\begin{aligned} w &::= u\,(v)^\omega \\ u &::= \varepsilon \mid 0 \mid 1 \mid 0 \bullet u \mid 1 \bullet u \\ v &::= 0 \mid 1 \mid 0 \bullet v \mid 1 \bullet v \end{aligned} \tag{63}$$

u is called the *prefix* of w, v is the *period* of w, and $(v)^\omega = \lim_n v^n$ denotes the infinite repetition of v. ε is the empty binary word. In order to avoid confusion between parentheses denoting periodic binary words and usual parentheses, the former are colored red. The associated ω symbol is also red.

For convenience, we adopt a power notion for repeated bits:

$$\begin{aligned} b^n &= b \bullet b^{n-1} \quad (b \in \mathbb{B}, n \in \mathbb{N}^\star) \\ b^0 &\triangleq \varepsilon \end{aligned} \tag{64}$$

A periodic binary word with an empty prefix is called a *strictly periodic binary word* ($w = (v)^\omega$).

A periodic binary word has infinitely many representations:

$$\text{Let } b \in \mathbb{B}, u, v \in \mathbb{B}^*, u \bullet b\,(v \bullet b)^\omega = u\,(b \bullet v)^\omega \tag{65}$$

Notation A.1 (Length of a binary word). $|w|$ denotes the length of the binary word w.

Notation A.2. $|w|_b$ denotes the number of bits set to $b \in \mathbb{B}$ in the binary word w.

Notation A.3. $w\,[k]$ denotes the k^{th} bit of the binary word w.

Notation A.4. $w[k..l]$ denotes the (sub) binary word from w starting at the k^{th} bit upto the l^{th} bit included.

Notation A.5. $w[k..]$ denotes the (sub) binary word from w starting at the k^{th} bit. Possibly infinite.

A.2 Operations on binary words

Definition A.5 (Number of 1 upto k)**.** $w \downarrow k$ denotes the number of 1 upto the k^{th} bit included in the binary word w.

$$w \downarrow k \triangleq |w[1..k]|_1 \tag{66}$$

$$\begin{aligned} &\text{Let } k \in \mathbb{N}^\star, b \in \mathbb{B}, w \text{ a binary word} \\ &\qquad w \downarrow 0 \triangleq 0 \\ &\qquad b \bullet w \downarrow k = b + (w \downarrow (k-1)) \end{aligned} \tag{67}$$

Definition A.6 (Index of the k^{th} one)**.** $w \uparrow k$ denotes the index of the k^{th} one in the binary word w.

$$w \uparrow k \triangleq j \in \mathbb{N}^\star \text{ such that } w[j] = 1 \wedge (w \downarrow j = k) \tag{68}$$

$$\begin{aligned} &\text{Let } k \in \mathbb{N}^\star, w \text{ a binary word} \\ &\qquad w \uparrow 0 \triangleq 0 \\ &\qquad w \uparrow k \triangleq \omega \text{ if } |w|_1 < k \\ &\qquad (1 \bullet w) \uparrow k = 1 + w \uparrow (k-1) \\ &\qquad (0 \bullet w) \uparrow k = 1 + w \uparrow k \end{aligned} \tag{69}$$

Definition A.7 (Binary word composition)**.** For any two binary words w_1 and w_2, the binary word composition ($\circ$ operator) is defined as follows:

$$\begin{aligned} (0 \bullet w_1) \circ w_2 &= 0 \bullet (w_1 \circ w_2) \\ (1 \bullet w_1) \circ (b \bullet w_2) &= b \bullet (w_1 \circ w_2) \ \text{(for } b \in \mathbb{B}) \\ (0 \bullet w_1) \circ \varepsilon &= 0 \bullet (w_1 \circ \varepsilon) \\ (1 \bullet w_1) \circ \varepsilon &= \varepsilon \\ \varepsilon \circ w_2 &= \varepsilon \end{aligned} \tag{70}$$

Properties:

$$|w_1 \circ w_2| = \min\{|w_1|, w_1 \uparrow (|w_2| + 1) - 1\} \tag{71}$$

Definition A.8 (Binary word union)**.** For any two binary words w_1 and w_2, the binary word addition ($+$ operator) is defined as follows:

$$\begin{aligned} (b_1 \bullet w_1) + (b_2 \bullet w_2) &= (b_1 \text{ or } b_2) \bullet (w_1 + w_2) \\ \varepsilon + w &= w \\ w + \varepsilon &= w \\ \varepsilon + \varepsilon &= \varepsilon \end{aligned} \tag{72}$$

Definition A.9 (Binary word shift left)**.** For any binary word w, the binary word shift left ($\ll$ operator) is defined as follows:

$$\begin{aligned} \varepsilon \ll 1 &= \varepsilon \\ (b.w) \ll 1 &= w \end{aligned} \tag{73}$$

B Observer specifications

The detection of a violation by an observer is considered as an error, and the analysis is stopped. So, there is no need for the violation state; emitting the output signal V is sufficient. In the FSM given in this annex, the sink state has been removed, and for each violation transition, its source and target states are the same. A further simplification consists in removing any transition having the same source and target state, but not emitting the violation signal. For instance in figure 34, the transition from S0 to itself, and labeled by a.b has been omitted.

For all state machines contained in this section, all non explicit variable expressions trigger a loop transition (same source and target state).

B.1 Alternation

Strict alternation

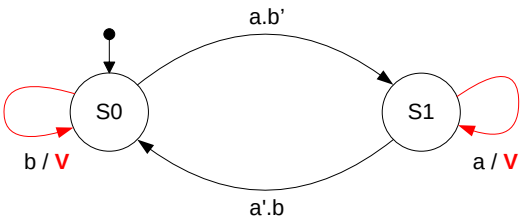
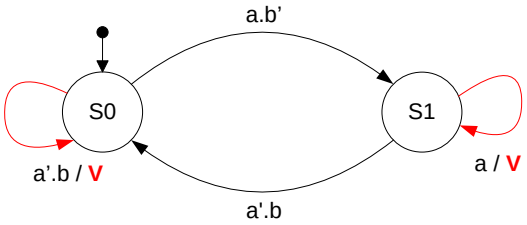


Figure 33: FSM detecting the violation of the strict alternation relation.

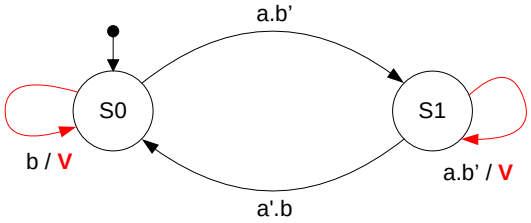
Right non strict alternation



$$\left(k \in \mathbb{N}^*\right)\ a[k] \preccurlyeq b[k] \prec a[k+1] \text{ (rnsalt)}$$

Figure 34: FSM detecting the violation of the right non strict alternation relation.

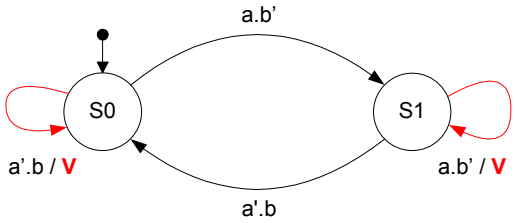
Left non strict alternation



$$\left(k \in \mathbb{N}^*\right)\ a[k] \prec b[k] \preccurlyeq a[k+1] \text{ (lnsalt)}$$

Figure 35: FSM detecting the violation of the left non strict alternation relation.

Non strict alternation

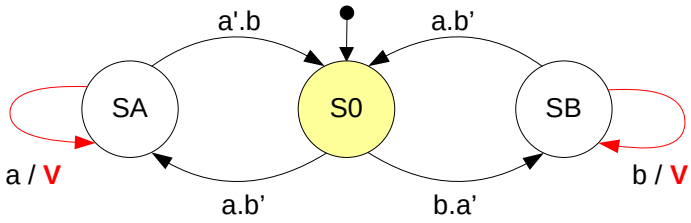


$$\left(k \in \mathbb{N}^*\right)\ a[k] \preccurlyeq b[k] \preccurlyeq a[k+1] \text{ (nsalt)}$$

Figure 36: FSM detecting the violation of the non strict alternation relation.

B.2 Synchronization

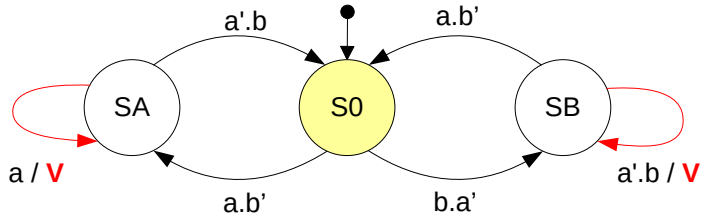
Strict synchronization



$$\left(k \in \mathbb{N}^*\right)\left(a[k] \prec b[k+1]\right) \wedge \left(b[k] \prec a[k+1]\right)$$

Figure 37: FSM detecting the violation of the strict synchronization relation.

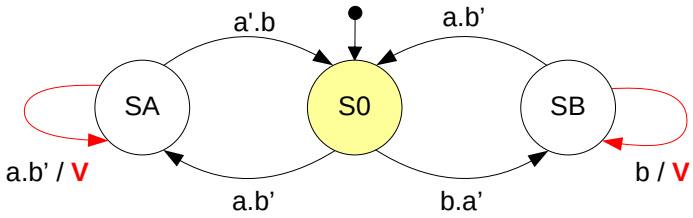
Right non strict synchronization



$$\left(k \in \mathbb{N}^*\right)\left(a[k] \preccurlyeq b[k+1]\right) \wedge \left(b[k] \prec a[k+1]\right)$$

Figure 38: FSM detecting the violation of the right non strict synchronization relation.

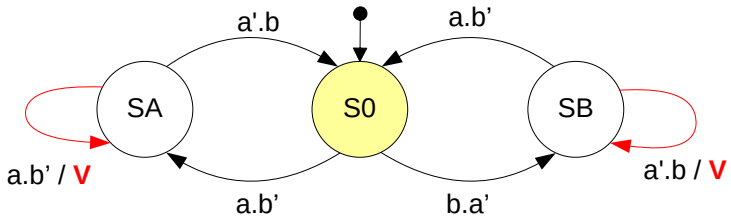
Left non strict synchronization



$$\left(k \in \mathbb{N}^*\right)\left(a[k] \prec b[k+1]\right) \wedge \left(b[k] \preccurlyeq a[k+1]\right)$$

Figure 39: FSM detecting the violation of the left non strict synchronization relation.

Non strict synchronization



$$\left(k \in \mathbb{N}^*\right)\left(a[k] \preccurlyeq b[k+1]\right) \wedge \left(b[k] \preccurlyeq a[k+1]\right)$$

Figure 40: FSM detecting the violation of the non strict synchronization relation.

B.3 By-packet Alternation

Strict alternation

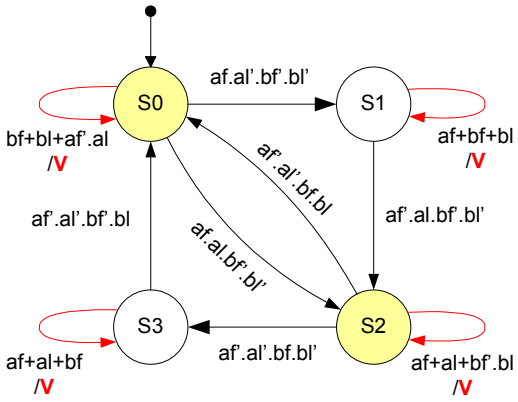


Figure 41: FSM detecting the violation of the by-packet strict alternation relation.

Right non strict alternation

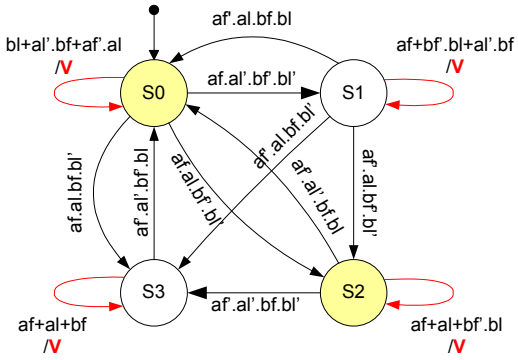


Figure 42: FSM detecting the violation of the by-packet right non strict alternation relation.

Left non strict alternation

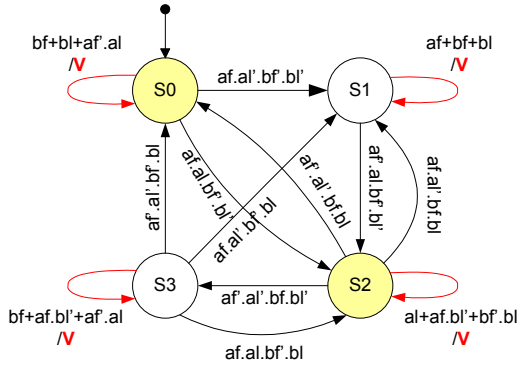


Figure 43: FSM detecting the violation of the by-packet left non strict alternation relation.

Non trict alternation

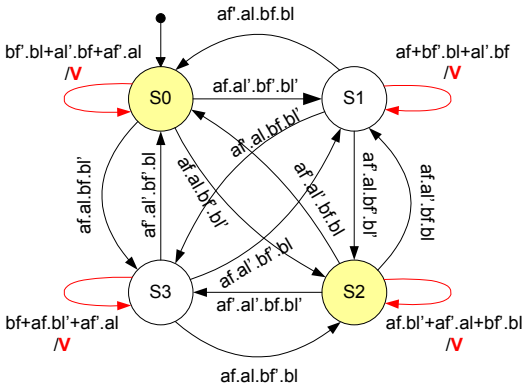


Figure 44: FSM detecting the violation of the by-packet non strict alternation relation.

Contents

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