Syntax and Semantics of the Clock Constraint Specification Language (CCSL)

Charles André

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Syntax and Semantics of the Clock Constraint Specification Language (CCSL)

Charles André

N° 6925 — version 2

version initiale Mai 2009 — version révisée Juin 2009

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6925--FR+ENG

Syntax and Semantics of the Clock Constraint Specification Language (CCSL)

Charles André*

Thème COM — Systèmes communicants
Projet Aoste

Rapport de recherche n° 6925 — version 2 — version initiale Mai 2009 — version révisée Juin 2009 — 37 pages

Abstract: The UML Profile for Modeling and Analysis of Real-Time and Embedded (MARTE) systems has recently been adopted by the OMG. Its Time Model extends the informal and simplistic Simple Time package proposed by UML2 and offers a broad range of capabilities required to model real-time systems. The MARTE OMG specification introduces a Time Structure inspired from Time models of the concurrency theory and proposes a new clock constraint specification language (CCSL) to specify, within the context of UML, logical and chronometric time constraints.

This report specifies the syntax and a formal semantics of a subset of CCSL, called *kernel* CCSL. This semantics is to be the reference semantics of CCSL.

Key-words: CCSL, syntax, semantics, time constraints, UML

* 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

Syntaxe et sémantique du langage de spécification des contraintes d'horloges (CCSL)

Résumé : Le profil UML pour la Modélisation et l'Analyse des systèmes Temps Réel et Embarqués (MARTE) a été récemment adopté par l'OMG. Son modèle de Temps étend le modèle simpliste défini dans le paquetage "Simple Time" de la spécification UML2 et offre des possibilités adaptées aux systèmes temps réel. La spécification OMG MARTE introduit uns Structure de Temps (Time Structure) inspirée des modèles de temps de la théorie du parallélisme et propose un langage de spécification de contraintes d'horloges appelé CCSL. Ce langage permet de spécifier dans le cadre d'UML, des contraintes aussi bien de temps chronométrique que de temps logique.

Ce rapport définit la syntaxe et la sémantique d'un sous-ensemble noyau de CCSL. La sémantique donnée servira de référence.

Mots-clés : CCSL, syntaxe, sémantique, contraintes temporelles, UML

1 Introduction

The *Unified Modeling Language* (UML) is a widely accepted language for system modeling (structural and behavioral aspects). However, when time plays a central role in the sytem behavior, UML is inadequate. Its model of time, called *Simple Time*, is too simple to address, for instance, real-time system time requirements. In its *Common Behavior* chapter, the UML specification [1] explicitly says that system modeling that demand advanced time concepts "will use a more sophisticated model of time provided by an appropriate profile". The UML Profile for *Modeling and Analysis of Real-Time and Embedded*' systems [2] (MARTE), is such a profile.

MARTE introduces a rich model of time (Chapter 9: Time Modeling) that supports dense and discrete time, chronometric and logical time, simple and multiple time references. In MARTE, a *Clock* is a model element giving access to the model time structure. When using the MARTE time model, several—interdependent— clocks are generally defined in a model. Clock mutual dependence can be specified with a dedicated language, called *Clock Constraint Specification Language* (CCSL). CCSL is defined in annex C3 of the UML specification. The syntax of CCSL is non normative and its semantics is informal (English description).

This report defines a *formal semantics* for a subset of CCSL, called *Kernel Clock Constraint Language* (abbreviated as KCCL). This semantics is to be the reference semantics of CCSL. It is applied to *execute* timed UML models in software environment like TimeSquare [3]. TimeSquare is a collection of plug-ins developped by the project team AOSTE to apply the MARTE time model, to specify clock constraints and to analyze them[1]. CCSL can also be used with non-UML models to specify constraints on event occurrences.

This report is organized as follows. Section 2 presents the meta-models: a simplified version of the MARTE time model (Sec 2.1) and the kernel clock constraint language (Sec. 2.2). Section 3 introduces the syntax of KCCL. A concise symbolic notation, inspired by processus algebra, is also proposed. A *structural operational semantics* of KCCL is then defined in Section 4. A KCCL specification is transformed into a set of logical constraints. Section 5 describes how to solve these constraints with a BDD-based Boolean solver. Different policies to select a solution from the many possible ones are then discussed. Finally, we conclude with existing success applications of CCSL and its forthcoming developments.

2 Metamodels

2.1 Time Model

We have defined the *Time Model* part of the UML profile for "Modeling and Analysis of Real-Time and Embedded" systems [2], (MARTE). Figure 1 shows a simplified version of the MARTE Time meta-model augmented with dynamic features. For a detailed description the reader is urged to refer to the MARTE specification.

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

2.1.1 Static view

The upper part of the figure represents the static model elements and their relationships. A *Time Domain* consists of one or many *clocks*. Each clock *clk* owns an ordered set of *instants* $\mathcal{I}_{clk}$. One of these instants is distinguished as the *current instant* of the clock.

The instants of the clocks are *a priori* independent. However, in most applications there exist dependence relationships between pairs of instants. Three instant relations have been introduced in MARTE: *coincidence* (denoted $\equiv$), *precedence* (denoted $\preccurlyeq$), and *exclusion* (denoted #). A fourth relation, the *strict precedence* (denoted $\prec$), is derived from the others ($\prec \triangleq \preccurlyeq \setminus \equiv$). These instant relations are more conveniently expressed by *clock constraints*, which are the topic of this report. Thus, a time domain also owns a (usually not empty) set of clock constraints.

2.1.2 Dynamic view

A time domain with its sets of clocks and clock constraints specifies a *time system*. A temporal evolution—*i.e.,* one execution of the time system or *run*—is a kind of non-sequential process [4] whose events are clock tickings. Thus, in a first approximation, a temporal evolution is a partially ordered set of instants. In fact, because of our coincidence relation, the partial ordering is not on instants but on sets of coincident instants [5].

A *configuration* is a set of coexisting instants, *i.e.,* a *slice* of the occurrence net [6] representing a run. Of course, since the instants of a clock are strictly ordered, a configuration cannot contain two instants of the same clock. When a clock *ticks* (the current instant of the clock moves to the next instant of this clock), the configuration changes. Several clocks can fire[2] *simultaneously*. A set of simultaneous clock firings is called a *step*. A particular execution of a time system (*run*) is a partially ordered set of steps.

The issue is to compute runs that meet all the clock constraints imposed in the time domain. This is done by a *constraint solver* applying a given *policy*.

Remark: The reader may wonder why considering the notion of *simultaneous* firings intead of *concurrent* firings. *Concurrency*, such as defined in the General net Theory (C.A. Petri), is a kind of independence relation (absence of causal ordering). In this theory, *coincidence* is a very strong relation that relates two events occurring at one space-time point (one place, one instant). Of course, two coincident events are not concurrent because they are tighly dependent. In fact, the two events *are* a unique event. In our approach of time, clock ticks—changes in the current instant of a clock—play the role of events (or more precisely, event occurrences). The idea to define a step as a set of concurrent clock firings is not adequate because this precludes coincident clock firings. For us, coincidence is not the unification of two instants: coincident instants keep their individuality, they are only forced to occur jointly. This constraint is not necessarily a causal condition. Most of the time, it is a mere design choice. That is the reason why we have adopted another adjective: *simultaneous*. A step (simultaneous clock firings) contains coincident firings that *must* be fired

[2] another word to say that the clock ticks.

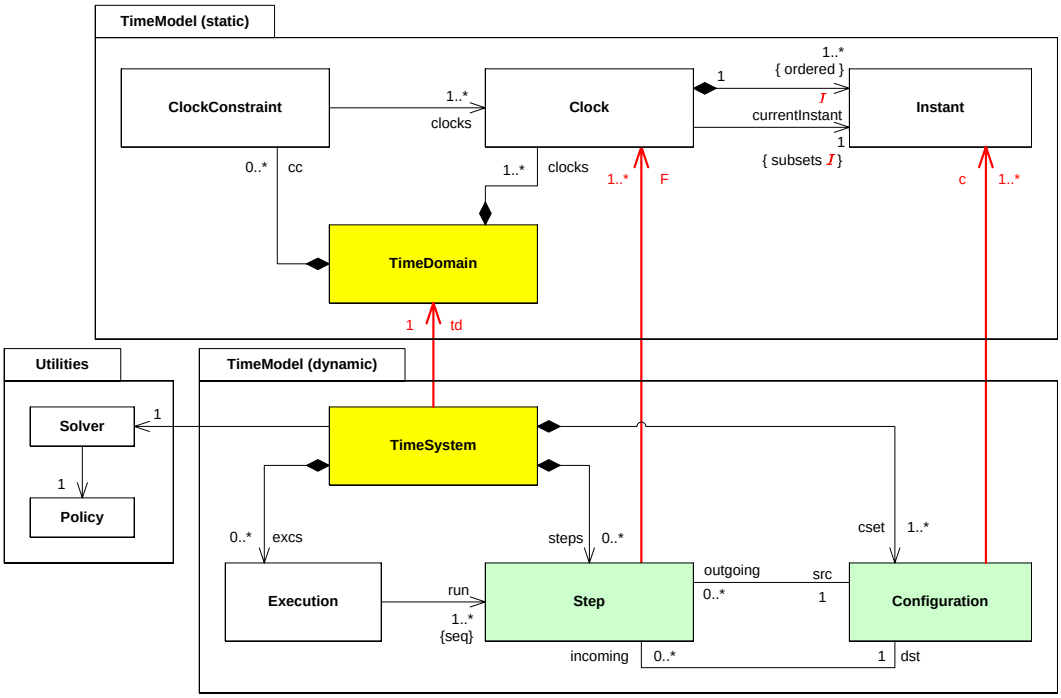

Figure 1: Static and Dynamic views of Time

altogether, and possible concurrent firings. *Simultaneity* is a central concept in *synchronous reactive* modeling. Since covering the concept of instants of synchronous reactive models is also part of the MARTE time model objectives, we have adopted the concept of *simultaneous firings*.

2.2 Clock constraints

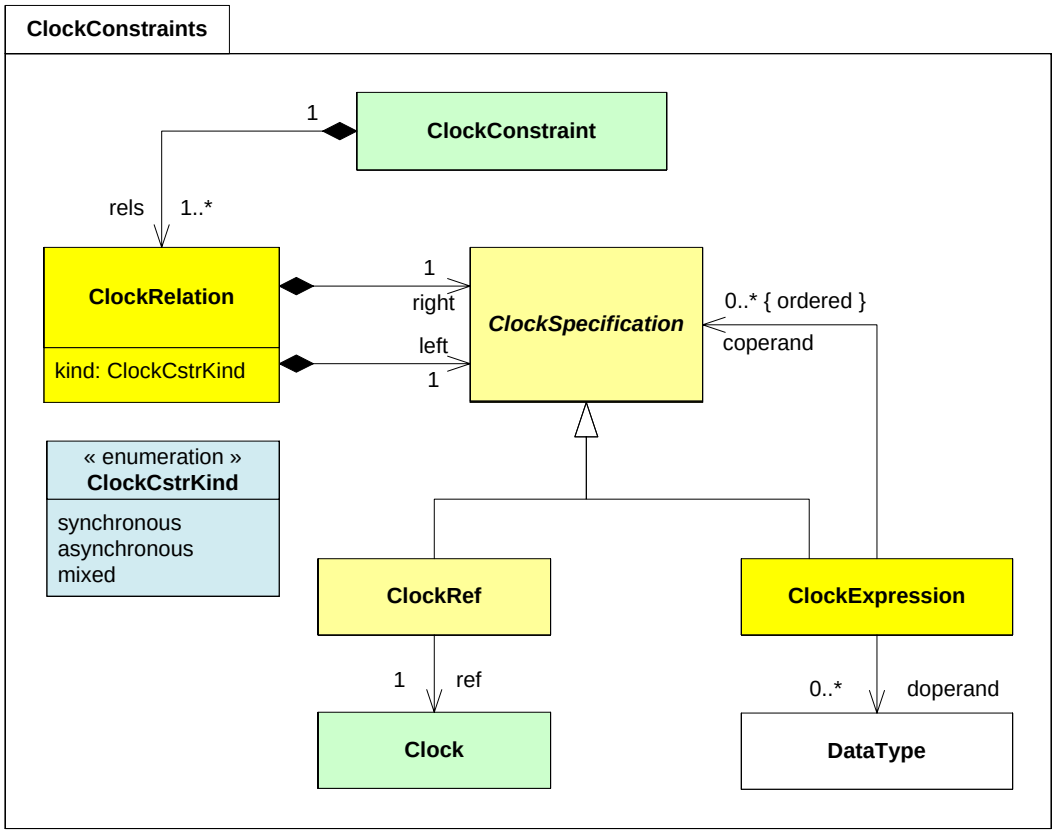

Figure 2: Clock constraint meta-model

The clock constraints of a time domain are specified in CCSL (Clock Constraint Specification Language). We have introduced this language in the MARTE specification. It is a non normative specification, and its semantics has only been given in an informal way (English specification). Beyond the MARTE specification, we have given a concrete syntax and a formal semantics to CCSL. A *kernel* CCSL, called Kernel Clock Constraint Language (KCCL) has also been defined. Using the primitives provided by this kernel, complex constraints can be elaborated. Another advantage of having a kernel is that it is sufficient to define the semantics of the kernel primitives to give the semantics of any CCSL specification.

The abstract syntax of CCSL is given in Fig. 2. A *clock constraint* consists of at least one *clock relation*. A clock relation relates two *clock specifications*. A clock specification can be either a simple reference to a clock or a *clock expression*. A clock expression refers to one or more clock specifications and possibly to additional operands. The attribute kind of a clock relation indicates if the relation is based on the coincidence relation (synchronous), the precedence relation (asynchronous), or a combination of both (mixed).

The meta-model in Fig. 2 is a simplified version. To describe the semantics of the clock constraints, we needs a more detailed meta-model (Fig. 3).

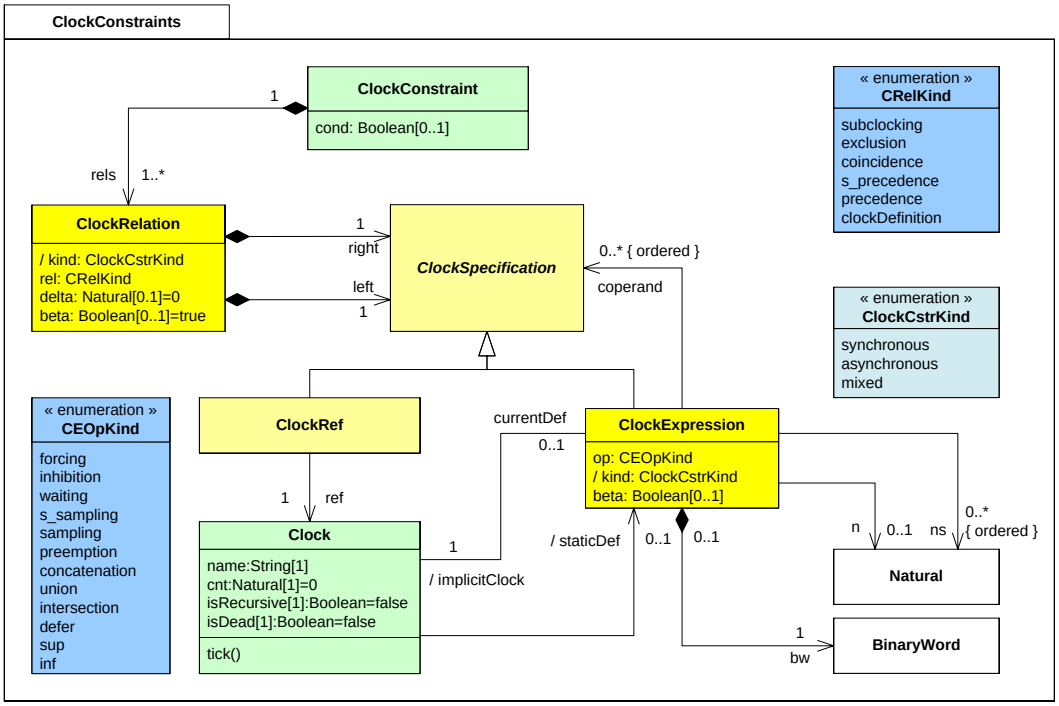

Figure 3: More detailed (Kernel) Clock constraint meta-model

The names of the clock relations and the clock expressions used in KCCL are defined in the enumerations CRelKind and CEOpKind. The attribute kind of the ClockRelation is now a derived attribute (see the derivation rules below). Attributes and operations have been added to the meta-model classes. They have been introduced to make the semantic rules easier. doperand has been redefined as bw whose type is BinaryWord, n whose type is Natural, and ns a sequence of natural numbers. The type BinaryWord is detailled in Annex A. For

now, the reader has only to know that a binary word is a, possibly infinite, sequence of bits (0 or 1). Derived properties are also introduced along with associated OCL rules.

```
context ClockExpression
inv:    if self.op = CEOpKind::waiting
        then    self.n->notEmpty()
        else    self.n->isEmpty()
        endif
inv:    if self.op = CEOpKind::defer
        then    self.ns->notEmpty()
        else    self.ns->isEmpty()
        endif
inv:    if self.op = CEOpKind::defer or
           self.op = CEOpKind::s_sampling or
           self.op = CEOpKind::sampling or
           self.op = CEOpKind::sup or
           self.op = CEOpKind::inf
        then    self.kind = ClockCstrKind::mixed
        else    self.kind = ClockCstrKind::synchronous
        endif

context ClockRelation
inv:    self.rel = CRelKind::clockDef implies
           self.left.oclIsTypeOf(ClockRef) and
           self.right.oclIsTypeOf(ClockExpression) and
           self.left.staticDef->notEmpty() and
           self.left.currentDef->notEmpty() and
           self.left.staticDef = self.right and
           self.right.implicitClock = self.left
inv:    if self.rel = CRelKind::s_precedence or
           self.rel = CRelKind::precedence
        then    self.delta->notEmpty()
        else    self.delta->isEmpty()
        endif
inv:    if self.rel = CRelKind::s_precedence or
           self.rel = CRelKind::exclusion
        then    self.kind = ClockCstrKind::asynchronous
        else    if self.rel = CRelKind::precedence
                then self.kind = mixed
                else self.kind = synchronous
                endif
        endif
```

3 Syntax of the kernel CCSL

This section defines the syntax of the kernel clock constraint language on a set of clocks $\mathcal{C}$ (denoted KCCL $_{\mathcal{C}}$, or simply KCCL when $\mathcal{C}$ is understood). With the primitive constructs provided by the kernel, new constructs can be derived and proposed in libraries. For the sake of conciseness, we use a symbolic notation for the kernel operators and constructs. There also exists a concrete textual syntax given in Table 6. The reader should not focus on this concrete syntax since CCSL specifications are usually given interactively through dialog boxes offered by TimeSquare.

3.1 Kernel relations and expressions

3.1.1 Notation

In what follows, grammatical rules are given in the BNF notation. ::= means "is defined by"; | means "or"; non terminal symbols are written in a sans-serif font (*e.g.*, CC, natural). Terminal symbols are in red fonts (*e.g.*, |, ⊂).

Table 1: Constraints and relations

CC	::=		(clock constraint)
		CC | CR	(parallel composition)
	|	CR	
	|	CR if bool	(conditional constraint)
CR	::=		(clock relation)
		CS r_{op} CS	

Continued on next page

Table 2: Relation operators

r_{op}	::=		(relation operator)
		⊂	(subclocking)
	|	#	(exclusion)
	|	=	(coincidence)

Continued on next page

|	$\prec$	(s_precedence)
|	$\preccurlyeq$	(precedence)

Table 3: Clock specifications

CS	::=		(clock specification)
		clock	(clock reference)
	|	CE	
CE	::=		(clock expression)
		bool ? CE : CE	(conditional expression)
	|	clock	(clock reference)
	|	! 1	(force)
	|	! 0	(inhibit)
	|	CE^natural	(await)
	|	CE $\searrow$ CE	(s_sample)
	|	CE $\searrow$ CE	(sample)
	|	CE $\downarrow$ CE	(upto)
	|	CE $\bullet$ CE	(concat)
	|	CE + CE	(union)
	|	CE $*$ CE	(inter)
	|	CE(naturalSequence) $\rightsquigarrow$ CE	(defer)
	|	CE $\vee$ CE	(sup)
	|	CE $\wedge$ CE	(inf)

The non terminal clock is an identifier referencing a clock of $\mathcal{C}$. bool stands for a Boolean expression. natural is an integer expression evaluating to a non negative integer value. naturalSequence is a sequence of natural.

Recursion The clock attribute isRecursive is a Boolean set to true for clock defined by a relation of the form clock $=$ CE $\bullet$ clock, where clock denotes the same clock on both sides of the clock relation. In this case, the property staticDef is set to CE. The following OCL constraint holds:

```
context Clock
inv:    if self.isRecursive then
                self.staticDef->notEmpty() and
                self = self.staticDef.implicitClock
        else
                self.staticDef->isEmpty()
```

endif

3.2 Simple KCCL

The above syntax allows nested clock expressions. In order to make the specification of the semantics of KCCL easier, we propose now a simpler syntax forbidding nested clock expressions. This entails no loss of generality because nested clock expressions are replaced by clock definitions, one per clock expression. The new syntax in given in Tables 4 and 5. The non-terminals are similar to the ones used in the original syntax, prefixed by an 'S', short for Simple. The clock relation "clock definition" is a simple clock relation which associates a clock with a simple clock expression. Of course, $\mathcal{C}$ includes the newly introduced clocks. Notice that in Simple KCCL, a clock relation applies to two clock references instead of two clock specifications.

Table 4: Simple constraints and relations

SCC	::=		(clock constraint)
		SCC | SCR	(parallel composition)
	|	SCR	
	|	SCR if bool	(conditional constraint)
SCR	::=		(clock relation)
		clock r_{op} clock	
	|	clock $\triangleq$ SCE	(clock definition)

Continued on next page

Table 5: Simple clock expressions

SCE	::=		(simple clock expression)
		bool ? clock : clock	(conditional expression)
	|	clock	(clock reference)
	|	! 1	(force)
	|	! 0	(inhibit)
	|	clock^natural	(await)
	|	clock ↘ clock	(s_sample)
	|	clock ↘ clock	(sample)
	|	clock ↯ clock	(upto)
	|	clock • clock	(concat)

Continued on next page

	clock + clock	(union)
	clock ∗ clock	(inter)
	clock(naturalSequence) ⇝ clock	(defer)
	clock ∨ clock	(sup)
	clock ∧ clock	(inf)

3.3 Transformation KCCL to Simple KCCL

This transformation is specified by conditional rewriting rules of two kinds: clock constraint rewriting rules (denoted →), and clock specification rewriting rules (denoted ↠).

$$\rightarrow : \mathsf{CC} \rightarrow Set(\mathsf{SCR}) \tag{1}$$
$$\twoheadrightarrow : \mathsf{CS} \rightarrow \mathsf{clock} \times Set(\mathsf{SCR}) \tag{2}$$

In both rules, $Set(\mathsf{SCR})$ is the set of simple clock relations to be substituted to the left-hand side term. In the second rule, the clock result is a reference to a clock. As shown in the kernel clock constraint meta-model (Fig. 3), this clock is either a given clock (ref property of a ClockRef) or an implicit clock, created during the transformation and attached to a clock expression.

3.3.1 Clock relation transformations

$$\frac{cr \twoheadrightarrow scrs}{cr \text{ if } \beta \rightarrow \beta \ ? \ scrs : \emptyset} \quad \text{(cond. rel. transf.)} \tag{3}$$

In rule 3, the conditional operator is applied to sets: $\beta \ ? \ scrs : \emptyset$ is $scrs$ (a set of simple clock relations) if $\beta = 1$, and the empty set otherwise.

$$\frac{\begin{array}{c} cs_1 \twoheadrightarrow c_1, cds_1 \\ cs_2 \twoheadrightarrow c_2, cds_2 \end{array}}{cs_1 \ \mathsf{r_{op}} \ cs_2 \rightarrow \{c_1 \ \mathsf{r_{op}} \ c_2\} \cup cds_1 \cup cds_2} \quad \text{(rel. transf.)} \tag{4}$$

cds_1 (cds_2, respectively) is a set of clock definitions generated while transforming clock specification cs_1 (cs_2, respectively).

3.3.2 Clock expression transformations

$$c \twoheadrightarrow c, \emptyset \quad \text{(clock ref. transf.)} \tag{5}$$

Rule 5 applies when the clock specification is a ClockRef (meta-model Fig. 3). In this case, there is no actual change.

0-ary clock expression transformations:

$$! \, 1 \twoheadrightarrow c = \text{new clock}, \{c \boxed{\triangleq} \, !1\} \quad \text{(clock forcing transf.)} \tag{6}$$

$$! \, 0 \twoheadrightarrow c = \text{new clock}, \{c \boxed{\triangleq} \, !0\} \quad \text{(clock inhib. transf.)} \tag{7}$$

Unary clock expression transformation:

$$\frac{ce_1 \twoheadrightarrow c_1, cds_1}{ce\hat{}n \twoheadrightarrow c = \text{new clock}, \{c \boxed{\triangleq} \; c_1\hat{}n\} \cup cds_1} \quad \text{(await transf.)} \tag{8}$$

Binary clock expression transformations:

$$\frac{\begin{array}{c} ce_1 \twoheadrightarrow c_1, cds_1 \\ ce_2 \twoheadrightarrow c_2, cds_2 \end{array}}{ce_1 \; \mathsf{e}_{\mathsf{op2}} \; ce_2 \twoheadrightarrow c = \text{new clock}, \{c \boxed{\triangleq} \; c_1 \; \mathsf{e}_{\mathsf{op2}} \; c_2\} \cup cds_1 \cup cds_2} \quad \text{(bin. expr. transf.)} \tag{9}$$

The binary clock expression operators $\mathsf{e}_{\mathsf{op2}}$ are $\searrow$, $\not\searrow$, $\lightning$, $\bullet$, $+$, $*$, $\rightsquigarrow$, $\vee$, and $\wedge$.

Conditional clock specification transformation:

$$\frac{\begin{array}{c} cs_1 \twoheadrightarrow c_1, cds_1 \\ cs_2 \twoheadrightarrow c_2, cds_2 \end{array}}{\beta \; ? \; cs_1 : cs_2 \twoheadrightarrow c = \text{new clock}, \{c \boxed{\triangleq} \; \beta \; ? \; c_1 : c_2\} \cup (\beta \; ? \; cds_1 : cds_2)} \quad \text{(cond. spec. transf.)} \tag{10}$$

3.4 Example of transformation

Let $(a + b) \lightning c \boxed{=} d$ be a KCCL specification consisting of one clock relation only. Given clocks are a, b, c, and d. Rule 4 applies and demands the transformation of the two clock specifications $(a + b) \lightning c$ and d. Now, rule 9 applied to $(a + b) \lightning c$ creates a clock c_1, and requires transformations of clock expressions $a + b$ on the one hand, and c on the other hand.

The transformation of $a + b$ can be written as:

$$\frac{\begin{array}{c} a \twoheadrightarrow a, \emptyset \\ b \twoheadrightarrow b, \emptyset \end{array}}{a + b \twoheadrightarrow c_2 = \text{new clock}, \left\{ c_2 \boxed{\triangleq} \; a + b \right\}} \tag{11}$$

The transformation of c is trivial:

$$c \twoheadrightarrow c, \emptyset \tag{12}$$

From Eqs. 11 and 12, we can complete the transformation of $(a + b) \lightning c$:

$$\frac{\begin{array}{c} a + b \twoheadrightarrow c_2, \left\{ c_2 \boxed{\triangleq} \ a + b \right\} \\ c \twoheadrightarrow c, \emptyset \end{array}}{a + b \twoheadrightarrow c_1, \left\{ c_1 \boxed{\triangleq} \ c_2 \lightning c \right\} \cup \left\{ c_2 \boxed{\triangleq} \ a + b \right\}} \tag{13}$$

The transformation of the right-hand side of the relation is also trivial:

$$d \twoheadrightarrow d, \emptyset \tag{14}$$

Taking account of Eq. 13 and Eq. 14, the result of the transformation of the initial relation is:

$$\left\{ c_1 \boxed{=} \ d \right\} \cup \left\{ c_1 \boxed{\triangleq} \ c_2 \lightning c, c_2 \boxed{\triangleq} \ a + b \right\}$$

Hence, the equivalent simple KCCL specification is

$$c_2 \boxed{\triangleq} \ a + b \ \Big| \ c_1 \boxed{\triangleq} \ c_2 \lightning c \ \Big| \ c_1 \boxed{=} \ d$$

3.5 Summary

Table 6 gathers the elements of simple KCCL syntax along with their textual forms. In this table c is a clock reference (either a given clock or an implicit clock generated during the transformation).

Table 6: Syntax of the Simple Kernel Clock Constraint Language

Ref.	Textual syntax	Notation	Comments
		Relations	
R1	c1 isSubClockOf c2	$c_1 \boxed{\subset} c_2$	
R2	c1 # c2	$c_1 \boxed{\#} c_2$	
R3	c1 = c2	$c_1 \boxed{=} c_2$	
R4	c1 strictly precedes c2	$c_1 \boxed{\prec} c_2$	
R5	c1 precedes c2	$c_1 \boxed{\preccurlyeq} c_2$	

Continued on next page

Ref.	Textual syntax	Notation	Comments
R6	c clockDef e	$c \triangleq e$	Internal use (for simple KCCL)
R7	r if b	r if b	r:relation, b: Boolean
		Expressions	
E1	force	! 1	
E2	inhibit	! 0	
E3	await n c	$c\hat{}n$	n: natural number, $n > 0$
E4	c1 strictly sampled c2	$c_1 \searrow c_2$	
E5	c1 sampled c2	$c_1 \searrow c_2$	
E6	c1 upto c2	$c_1 \not\searrow c_2$	
E7	c1 followedBy c2	$c_1 \bullet c_2$	
E8	c1 clockUnion c2	$c_1 + c_2$	
E9	c1 clockInter c2	$c_1 * c_2$	
E10	c1 deferred c2 for ns	$c_1(ns) \rightsquigarrow c_2$	ns: sequence of non-0 natural numbers
E11	c1 lub c2	$c_1 \vee c_2$	
E12	c1 glb c2	$c_1 \wedge c_2$	
E13	if b then c1 else c2	b ? $c_1 : c_2$	b: Boolean

4 Semantics

The suggestion of representing temporal evolutions as non-sequential processes (Section 2.1) is not effective: they are infinite structures and they hide choices (conflicts). Instead, we propose to give KCCL an operational structural semantics that allows the effective construction of temporal evolutions. Note that we consider only clocks with a discrete set of instants.

4.1 Clock Model

A *Clock Model* $\mathcal{M} = \langle \mathcal{C}, \mathcal{S} \rangle$ consists of a finite set of discrete clocks $\mathcal{C}$, constrained by a KCCL$_\mathcal{C}$ specification $\mathcal{S}$. A clock model is a TimeDomain of the TimeModel (Fig 1), whose constraints are specified in CCSL.

A clock model is static, the next subsection deals with the dynamics of this model.

4.2 Time System

A *Time System* $\mathcal{TS} = \langle \mathcal{C}, \mathcal{S} \rangle, \chi^0$ consists of a Clock Model $\mathcal{M} = \langle \mathcal{C}, \mathcal{S} \rangle$, along with an initial *configuration* χ^0.

A configuration χ of $\mathcal{TS}$ is a tuple of natural numbers $\chi : \mathcal{C} \to \mathbb{N}$, where for any clock $c \in \mathcal{C}$, $\chi(c)$ is the current instant of clock c. The initial configuration χ^0 is such that $(\forall c \in \mathcal{C})\chi^0(c) = 0$.

A *run* r of $\mathcal{TS}$ is a, generally infinite, sequence of *steps*: $r = F_1.F_2.\cdots.F_n.\cdots$, where $F_k \subset \mathcal{C}$ for all k.

$F \subset \mathcal{C}$ is fireable in $\langle \mathcal{C}, \mathcal{S} \rangle$ at χ, if the firing of every clock in F satisfies the clock constraint $\mathcal{S}$ at χ. This is denoted by $\langle \mathcal{C}, \mathcal{S} \rangle, \chi \xrightarrow{F}$. The firing of F leads to a new configuration χ' and possibly new clock constraints $\mathcal{S}'$, denoted as shown in Eq. 15.

$$\langle \mathcal{C}, \mathcal{S} \rangle, \chi \xrightarrow{F} \langle \mathcal{C}, \mathcal{S}' \rangle, \chi' \tag{15}$$

In the new configuration, the current index of every fired clocks is incremented by 1 (Eq. 16).

$$\chi'(c) = \begin{cases} \chi(c) + 1 & \text{if } c \in F, \\ \chi(c) & \text{otherwise.} \end{cases} \tag{16}$$

The change in $\mathcal{S}$ is explained later.

A run r can be rewritten as

$$\langle \mathcal{C}, \mathcal{S}^0 \rangle, \chi^0 \xrightarrow{F_1} \langle \mathcal{C}, \mathcal{S}^1 \rangle, \chi^1 \xrightarrow{F_2} \cdots \langle \mathcal{C}, \mathcal{S}^{n-1} \rangle, \chi^{n-1} \xrightarrow{F_n} \langle \mathcal{C}, \mathcal{S}^n \rangle, \chi^n \cdots \quad \text{where } \mathcal{S}^0 = \mathcal{S} \tag{17}$$

The semantics of a clock constraint $\mathcal{S}$ expressed in $\text{KCCL}_{\mathcal{C}}$ is given as a Boolean expression on $\mathscr{C}$, a set of Boolean variables in bijection with $\mathcal{C}$.

$$\text{let } \pi : \mathcal{C} \to \mathscr{C} \text{ bijection, and } [\![\ ]\!] : \text{KCCL}_{\mathcal{C}} \to \mathscr{B}_{\mathscr{C}} \tag{18}$$

$[\![\ ]\!]$ is defined by structural rewriting rules. For convenience, we denote $\pi(c)$ by c, for all c in $\mathcal{C}$. $\mathsf{c} = 1$ means that c is fireable in $\langle \mathcal{C}, \mathcal{S} \rangle$ at χ. More generally,

$$(\forall f : \mathscr{C} \to \{0,1\})\, \langle \mathcal{C}, \mathcal{S} \rangle, \chi \xrightarrow{|f|} \Leftrightarrow [\![\mathcal{S}]\!](f) = 1 \tag{19}$$

In Eq. 19, f is a valuation of $\mathscr{C}$, $[\![\mathcal{S}]\!](f) = 1$ says that $[\![\mathcal{S}]\!]$ evaluates to 1 for the valuation f, and $|f| \triangleq \{c \in \mathcal{C} \mid f(\pi(c)) = 1\}$. The symbol $\triangleq$ means "is defined by".

The next two subsections detail structural transformations from $\text{KCCL}_{\mathcal{C}}$ to Boolean expressions on $\mathscr{C}$ (*i.e.*, $\mathscr{B}_{\mathscr{C}}$). In Boolean expressions, we use operators $\Rightarrow$ (implication), $=$ (equality), $\#$ (exclusion), and ite(, ,) (if ... then ... else ...) such that for any Boolean expression t_1, t_2, t_3:

$$\begin{aligned} t_1 \Rightarrow t_2 &\Leftrightarrow \neg t_1 \wedge t_2 \\ t_1 = t_2 &\Leftrightarrow (t_1 \wedge t_2) \vee (\neg t_1 \wedge \neg t_2) \\ t_1 \# t_2 &\Leftrightarrow \neg t_1 \vee \neg t_2 \\ \text{ite}(t_1, t_2, t_3) &\Leftrightarrow (t_1 \wedge t_2) \vee (\neg t_1 \wedge t_3) \end{aligned}$$

A first rule is given right now. This rule expresses the composition of clock relations: the parallel composition of clock relations is the conjunction of the associated Boolean expressions.

$$[\![\mathsf{SCR}_1 \mid \mathsf{SCR}_2]\!] = [\![\mathsf{SCR}_1]\!] \wedge [\![\mathsf{SCR}_2]\!] \quad (\text{ paral }) \tag{20}$$

4.3 Clock relations

4.3.1 Conditional clock relation

A Clock relation can be defined conditionally to some Boolean β. When β is false, $[\![\mathsf{SCR}\ \text{if}\ \beta]\!]$ is true, whatever the clock relation. Else, we have to compute the Boolean expression associated with the clock relation.

$$[\![\mathsf{SCR}\ \text{if}\ \beta]\!] = (\beta \Rightarrow [\![\mathsf{SCR}]\!]) \quad (\text{rcond}) \tag{21}$$

4.3.2 Index-independent clock relations

Sub-clocking c_1 is a subclock of c_2 (or c_2 is a superclock of c_1) means that each instant of c_1 must be coincident with an instant of c_2. In logical words this says that c_1 ticks only if c_2 ticks, hence the logical implication.

$$[\![c_1 \boxed{\subset} c_2]\!] = (\mathsf{c}_1 \Rightarrow \mathsf{c}_2) \quad (\text{subclock}) \tag{22}$$

Recall that $\mathsf{c} = b_c(c) = [\![c]\!]$.

Clock exclusion Two clocks c_1 and c_2 can be declared exclusive, that is, none of their instants are coincident, or equivalently, it is forbidden that both c_1 and c_2 tick at a configuration. This is expressed by the Boolean expression $\mathsf{c}_1 \# \mathsf{c}_2$ equivalent to $\neg(\mathsf{c}_1 \wedge \mathsf{c}_2)$ and $\neg\mathsf{c}_1 \vee \neg\mathsf{c}_2$.

$$\left[\!\!\left[c_1 \boxed{\#} c_2\right]\!\!\right] = (\mathsf{c}_1 \# \mathsf{c}_2) \quad (\text{excl}) \tag{23}$$

Clock equality This is a special case of subclocking, when there is a bijection between the sets of instants of the two clocks. The Boolean expression states that c_1 ticks if and only if c_2 ticks.

$$[\![c_1 \boxed{=} c_2]\!] = (\mathsf{c}_1 = \mathsf{c}_2) \quad (\text{coinc}) \tag{24}$$

Clock definition As explained in subsection 3.2, this clock relation operator is for internal use. The left-hand side clock ticks whenever the right-hand side clock expression ticks.

$$\left[\!\!\left[c \boxed{\triangleq} \mathsf{SCE}\right]\!\!\right] = (\mathsf{c} = [\![\mathsf{SCE}]\!]) \quad (\text{clockDefinition}) \tag{25}$$

4.3.3 Index-dependent clock relations

The next two clock relations depend on the current instant of the concerned clocks. More precisely they depend on the difference of indexes between the two clocks. Let $\delta \triangleq \chi(c_1) - \chi(c_2)$. δ is the optional attribute delta of ClockRelation introduced in the clock constraint metamodel (Fig 3).

Clock strict precedence $c_1 \boxed{\prec} c_2$ is read "c_1 strictly precedes c_2". This means that for any χ in a run of the Time System, $\chi(c_1) \geq \chi(c_2)$. This formulation is less intuitive than the following: for any natural number k, the k^{th} instant of c_1 strictly precedes the k^{th} instant of c_2. This precedence between instants explains that this relation is also read as "c_1 is strictly *faster than* c_2". According to this definition, c_1, which is the faster of the two clocks, is never constrained. As for c_2, it is constrained only when its index becomes equal to the index of c_1. Under such a circonstance, c_2 must not tick.

$$\frac{\beta \triangleq (\delta = 0)}{[\![c_1 \boxed{\prec} c_2]\!] = (\beta \Rightarrow \neg \mathsf{c}_2)} \quad (\text{ sprec }) \tag{26}$$

Another consequence of this rule is that the following invariant property holds:
Invariant: $\delta \geqslant 0$

Clock non-strict precedence The non-strict precedence relation is similar to the previous one. The sole difference is in the possibility for c_2 to tick when $\delta = 0$, provided that c_1 also ticks. Hence the Boolean expression involving two implications.

$$\frac{\beta \triangleq (\delta = 0)}{[\![c_1 \boxed{\preccurlyeq} c_2]\!] = (\beta \Rightarrow (\mathsf{c}_2 \Rightarrow \mathsf{c}_1))} \quad (\text{ prec }) \tag{27}$$

The invariant property on δ still holds:
Invariant: $\delta \geqslant 0$

4.4 Clock expressions

During a run, clock expressions may change. So, we introduce *conditional rewriting rules* for clock expressions. A rewriting is expressed as $\mathsf{SCE} \rightarrow \mathsf{SCE}'$ where SCE' replaces SCE after a firing which meets the condition. As shown in Fig. 3, Sec. 2.2, an expression may own a binary word *bw*. In this case, the rewriting rule takes the form $\mathsf{SCE}, w \rightarrow \mathsf{SCE}', w'$ where binary words are written if needed.

A simple clock expression has an associated *implicit clock* (Fig. 3). In the rules below, c stands for the clock associated with the current clock expression.

0 and **1** are two predefined clocks. The former never ticks, the latter always ticks.

4.4.1 Conditional clock expression

A conditional clock expression defines a clock that behaves either as a clock c_1 or as another clock c_2 according to the value taken by the Boolean β.

$$[\![\beta \ ? \ c_1 : c_2]\!] = \mathrm{ite}(\beta, \mathsf{c}_1, \mathsf{c}_2) \quad (\text{econd}) \tag{28}$$

4.4.2 Terminating clock expressions

Terminating clock expressions define finite clocks (*i.e.,* clocks that eventually die). These clock expressions are used to build more complex clock expressions, especially through the clock concatenation.

Forcing The forcing expression forces a tick at the current step, then dies.

$$[\![!1]\!] = 1 \quad (\text{force}) \tag{29}$$

$$!1 \rightarrow \mathbf{0} \quad (\text{RWforce}) \tag{30}$$

Inhibition The inhibition expression forbids a tick at the current step, then dies.

$$[\![!0]\!] = 0 \quad (\text{inhib}) \tag{31}$$

$$!0 \rightarrow \mathbf{0} \quad (\text{RWinhib}) \tag{32}$$

Awaiting The awaiting clock expression $c1\hat{}n$ ticks in coincidence with the next n^{th} strictly future tick of c_1, and then dies.

$$\frac{\beta \triangleq (n = 1)}{[\![c_1\hat{}n]\!] = (\beta \wedge \mathsf{c}_1)} \quad (\text{await}) \tag{33}$$

$$\frac{c_1 \in F}{c_1\hat{}1 \rightarrow \mathbf{0}} \quad (\text{RWawait1}) \tag{34}$$

$$\frac{c_1 \in F \quad n > 1}{c_1\hat{}n \rightarrow c_1\hat{}(n-1)} \quad (\text{RWawait2}) \tag{35}$$

Strict sampling Sampling clock expressions involve two clocks. The first is considered as a trigger and the second as a time base. The sampling expression ticks in coincidence with the tick of the base clock immediately following a tick of the trigger clock, and then dies. There exist two versions of the sampling: either the strict one (the coincident tick of the base clock is strictly after the trigger tick) or the non-strict one (the coincident tick of the base clock may be coincident with the trigger tick when this one is coincident with a base clock tick).

$$[\![c_1 \searrow c_2]\!] = 0 \quad (\text{ ssampl }) \tag{36}$$

$$\frac{c_1 \in F}{c_1 \searrow c_2 \rightarrow c_2\hat{}1} \quad (\text{ RWssampl1 }) \tag{37}$$

$$\frac{c_1 \notin F}{c_1 \searrow c_2 \rightarrow c_1 \searrow c_2} \quad (\text{ RWssampl2 }) \tag{38}$$

Non strict sampling

$$[\![c_1 \searrow c_2]\!] = (\mathsf{c}_1 \wedge \mathsf{c}_2) \quad (\text{ sampl }) \tag{39}$$

$$\frac{c_1 \in F \quad c_2 \in F}{c_1 \searrow c_2 \rightarrow \mathbf{0}} \quad (\text{ RWsampl1 }) \tag{40}$$

$$\frac{c_1 \in F \quad c_2 \notin F}{c_1 \searrow c_2 \rightarrow c_1\hat{}1} \quad (\text{ RWsampl2 }) \tag{41}$$

Preemption The premption expression $c_1 \not\downarrow c_2$ behaves like c_1 while c_2 does not tick. When c_2 ticks, the expression dies.

$$[\![c_1 \not\downarrow c_2]\!] = (\mathsf{c}_1 \wedge \neg\mathsf{c}_2) \quad (\text{ upto }) \tag{42}$$

$$\frac{c_2 \in F}{c_1 \not\downarrow c_2 \rightarrow \mathbf{0}} \quad (\text{ RWupto }) \tag{43}$$

4.4.3 Non-terminating index-independent clock expressions

These clock expressions contrast with the terminating ones: they don't have explicit death. Among them, many are index-independent (*i.e.*, the rules do not refer to χ). The first three are special cases.

Clock reference This clock expression only states that a clock expression degenerated to a single clock behaves like this clock.

$$[\![c]\!] = \mathsf{c} \quad (\text{ ref }) \tag{44}$$

1 ticks at each step.

$$[\![\mathbf{1}]\!] = 1 \quad (\text{ always }) \tag{45}$$

0 never ticks.

$$[\![\mathbf{0}]\!] = 0 \quad (\text{ never }) \tag{46}$$

Clock concatenation The concatenation clock expression $c_1 \bullet c_2$ behaves like c_1 up to the death of c_1. When c_1 dies, the expression behaves like c_2. The concatenation may induce recursive definitions.

$$[\![c_1 \bullet c_2]\!] = \mathsf{c}_1 \quad (\text{ concat }) \tag{47}$$

$$\frac{\begin{array}{c} c \neq c_2 \\ c_1 \rightarrow \mathbf{0} \end{array}}{c_1 \bullet c_2 \rightarrow c_2} \quad (\text{ RWconcat }) \tag{48}$$

$$\frac{c_1 \rightarrow \mathbf{0}}{c_1 \bullet c \rightarrow c.staticDef \bullet c} \quad (\text{ RWrecur }) \tag{49}$$

Clock union The union clock expression $c_1 + c_2$ ticks whenever c_1 or c_2 ticks.

$$[\![c_1 + c_2]\!] = (\mathsf{c}_1 \vee \mathsf{c}_2) \quad (\text{ union }) \tag{50}$$

Clock intersection The intersection clock expression $c_1 * c_2$ ticks whenever both c_1 and c_2 tick.

$$[\![c_1 * c_2]\!] = (\mathsf{c}_1 \wedge \mathsf{c}_2) \quad (\text{ inter }) \tag{51}$$

Clock delay The delay clock expression $c_1(ns) \rightsquigarrow c_2$ is a rather complex expression involving two clocks (c_1, c_2) and a sequence of natural numbers (ns). c_1 is a trigger, c_2 a base clock. At each tick of c_1 the head of ns is dequeued and encoded in a binary word associated with the expression (bw, introduced in Fig. 3). This binary word is a kind of "diary" that contains the future rendez-vous with c_2 ticks. For instance, when c_1 ticks and the head of ns is 5, then the expression is expected to tick in coincidence with the next 5^{th} tick of c_2. Note that rendez-vous are not necessarily taken in a monotonic increasing order.

$$\frac{\beta \triangleq (bw = 1.w)}{[\![c_1(ns) \rightsquigarrow c_2]\!] = (\beta \wedge \mathsf{c}_2)} \quad (\text{ defer }) \tag{52}$$

$$\frac{c_1 \notin F \quad c_2 \in F \quad b \in \{0,1\}}{c_1(ns) \rightsquigarrow c_2, b.w \rightarrow c_1(ns) \rightsquigarrow c_2, w} \quad (\text{ RWdefer1 }) \tag{53}$$

$$\frac{c_1 \in F \quad c_2 \notin F \quad h \in \mathbb{N}^\star}{c_1(h.s) \rightsquigarrow c_2, w \rightarrow c_1(s) \rightsquigarrow c_2, w + (0^{h-1}.1)} \quad (\text{ RWdefer2 }) \tag{54}$$

$$\frac{c_1 \in F \quad c_2 \in F \quad b \in \{0,1\} \quad h \in \mathbb{N}^\star}{c_1(h.s) \rightsquigarrow c_2, b.w \rightarrow c_1(s) \rightsquigarrow c_2, w + (0^{h-1}.1)} \quad (\text{ RWdefer3 }) \tag{55}$$

4.4.4 Non-terminating index-dependent clock expressions

The last two clock expressions depend on χ, or more precisely on the difference of indexes between two clocks.

The fastest of slower clocks The sup clock expression $c_1 \vee c_2$ defines a clock that is slower than both c_1 and c_2 and whose k^{th} tick is coincident with the later of the k^{th} tick of c_1 and c_2.

$$\frac{\begin{array}{c}\beta_1 \triangleq (\chi(c_1) = \chi(c)) \\ \beta_2 \triangleq (\chi(c_2) = \chi(c))\end{array}}{[\![c_1 \vee c_2]\!] = \big((\beta_1 \Rightarrow c_1) \wedge (\beta_2 \Rightarrow c_2)\big)} \quad (\text{ sup }) \tag{56}$$

Invariant: $\chi(c) = \min\{\chi(c_1), \chi(c_2)\}$ and $\beta_1 \vee \beta_2 = 1$

The slowest of faster clocks This expression is the dual of the previous one. The inf clock expression $c_1 \wedge c_2$ defines a clock that is faster than both c_1 and c_2 and whose k^{th} tick is coincident with the earlier of the k^{th} tick of c_1 and c_2.

$$\frac{\begin{array}{c}\beta_1 \triangleq (\chi(c) = \chi(c_1)) \\ \beta_2 \triangleq (\chi(c) = \chi(c_2))\end{array}}{[\![c_1 \wedge c_2]\!] = \big((\beta_1 \Rightarrow c_1) \vee (\beta_2 \Rightarrow c_2)\big)} \quad (\text{inf}) \tag{57}$$

Invariant: $\chi(c) = \max\{\chi(c_1), \chi(c_2)\}$ and $\beta_1 \vee \beta_2 = 1$

5 Effective computation of Steps

In the previous section, a KCCL$_{\mathcal{C}}$ specification $\mathcal{S}$ has been transformed into a Boolean expression $[\![\mathcal{S}]\!]$. Now, we explain how to determine a set of clocks F to fire for a given configuration χ.

We proceed in steps:

1. (implicit) construction of the logically correct solutions of $[\![\mathcal{S}]\!]$. This is performed by a BDD-based solver;
2. choose one of these solutions according to the adopted policy;
3. effectively fire the clocks in F and update the state accordingly.

Step 1 is deterministic but usually yields several solutions. Step 2 makes a selection that can be non-deterministic. Step 3 applies the rewriting rules of the semantics, so that the clock sytem moves to a new state.

To illustrate this process, we consider a simple case, used as a running example.

5.1 Logical solutions

5.1.1 Example

Given $\mathcal{C} = \{a, b, c, d, e, f\}$, we consider the following Boolean expression

$$[\![\mathcal{S}]\!] = (d \Rightarrow b)|(b \Rightarrow a)|(c = e)|(c \Rightarrow a)|(b\#c) \tag{58}$$

So, $[\![\mathcal{S}]\!] = (\neg d \vee b) \wedge (\neg b \vee a) \wedge \big((c \wedge e) \vee (\neg c \wedge \neg e)\big) \wedge (\neg c \vee a) \wedge (\neg b \vee \neg c)$. Note that f is not effectively used in $[\![\mathcal{S}]\!]$.

5.1.2 Set of solutions

Let $\mathscr{V}$ the kernel of the Boolean expression $[\![\mathcal{S}]\!]$ defined in Eq. 59

$$\mathscr{V} \triangleq \{v : \mathscr{C} \to \{0,1\} | \, [\![\mathcal{S}]\!](v) = 1\} \tag{59}$$

Example : taking the variable ordering a, b, c, d, e, f
$\mathscr{V} = \{110{-}0{-}, 10101{-}, {-}0000{-}\}$ using *don't care* values, or explicitly $\mathscr{V} = \{110000, 110001, 110100, 110101, 101010, 101011, 000000, 000001, 100000, 100001\}$

5.1.3 Auxiliary sets

With any valuation $v \in \mathscr{V}$, we associate a subset $|v|$ of $\mathcal{C}$—subset of fireable clocks at χ—such that

$$|v| \triangleq \{c \in \mathcal{C} \mid v(\pi(c)) = 1\} \text{ where } \pi : \mathcal{C} \to \mathscr{C} \text{ is bijective} \tag{60}$$

Example : for $v = 110001$, $|v| = \{a, b, f\}$.

The set of fireable subsets of clocks at χ is $\mathcal{F}$ defined in Eq. 61.

$$\mathcal{F} \triangleq \{|v| \mid v \in \mathscr{V}\} \tag{61}$$

Example : $\mathcal{F} = \big\{\{a, b\}, \{a, b, f\}, \{a, b, d\}, \{a, b, d, f\}, \{a, c, e\}, \{a, c, e, f\}, \emptyset, \{f\}, \{a\}, \{a, f\}\big\}$.

We also introduce the set of fireable subsets of clocks including a given clock:

$$\mathcal{F}^c \triangleq \Big\{F \in \mathcal{F} \mid c \in F\Big\} \text{ for } c \in \mathcal{C} \tag{62}$$

Example : $\mathcal{F}^b = \big\{\{a, b\}, \{a, b, f\}, \{a, b, d\}, \{a, b, d, f\}\big\}$.

5.2 Subsets of clocks

5.2.1 Sets of enabled/disabled clocks

Given a time system $\langle \mathcal{C}, \mathcal{S} \rangle$ at χ, the subset of *disabled clocks* $D \subseteq \mathcal{C}$, and the subset of *enabled clocks* $E \subseteq \mathcal{C}$ at χ are such that

$$D = \{c \in \mathcal{C} \mid \mathcal{F}^c = \emptyset\} \tag{63}$$
$$E = \mathcal{C} \setminus D \tag{64}$$

Example : $D = \emptyset$; $E = \{a, b, c, d, e, f\}$

5.2.2 Required clocks

Given a clock $c \in \mathcal{C}$, a clock c' is said to be required for c in $\langle \mathcal{C}, \mathcal{S} \rangle$ at χ if for any solution, c in the solution implies c' in the same solution. c requires c' at χ is denoted $c \rightarrow_\chi c'$

$$(c \rightarrow_\chi c') \Leftrightarrow \Big(\mathcal{F}^c \subseteq \mathcal{F}^{c'}\Big) \tag{65}$$

Example : $b \rightarrow_\chi a$ because $\mathcal{F}^b = \{110000, 110001, 110100, 110101\}$; $\mathcal{F}^a = \{110000, 110001, 110100, 110101, 101010, 101011, 100000, 100001\}$, hence $\mathcal{F}^b \subset \mathcal{F}^a$. Other cases: $d \rightarrow_\chi b$, $c \rightarrow_\chi a$, $d \rightarrow_\chi a$, $e \rightarrow_\chi a$, $c \rightarrow_\chi e$, $e \rightarrow_\chi c$.

5.2.3 Set of fired clocks

All the clocks in E are not necessarily simultaneously fireable. For instance b and c are in E, but there is no valuation v such that $v(b) = v(c) = 1$ (no wonder since b and c have been specified exclusive).

This raises the issue of choosing a *consistent* subset F of E. By consistent we mean:

1. F contains only simultaneously fireable clocks;
2. if c is in F then all its required clocks are also in F.

In mathematical terms:

$$F \text{ is a consistent set iff } \begin{array}{l} F \in \mathcal{F} \wedge (\forall c' \in E) \\ \qquad (c \in F) \wedge (c \rightarrow_\chi c') \Rightarrow (c' \in F) \end{array}$$

This consistency criterium makes room for possibly many solutions. Several policies are predefined:

minimal: $F_{\min}$ is consistent and minimal: $\forall F' \in \mathcal{F}$, $F' \subseteq F_{\min} \Rightarrow F' = F_{\min}$.

maximal: $F_{\max}$ is consistent and maximal: $\forall F' \in \mathcal{F}$, $F_{\max} \subseteq F' \Rightarrow F' = F_{\max}$.

randomCausal: choose any c in E, and then build F that contains c, all its required clocks, and no other clocks (Given $c \in E$, $F = \{c\} \cup \{c' \in E \mid c \rightarrow_\chi c'\}$).

The last policy is called randomCausal because one enabled clock c is selected at random, and then other enabled clocks are added if they are causally linked to c (*i.e.,* involved with c in Boolean expressions like $c \Rightarrow c'$ or $c = c'$, for some c'). A different random policy is proposed later.

Example : possible fireable sets are

minimal $\{a\}$ or $\{f\}$.

maximal $\{a, b, d, f\}$ or $\{a, c, e, f\}$.

randomCausal for

a: $F = \{a\}$
b: $F = \{a, b\}$

c: $F = \{a, c, e\}$
d: $F = \{a, b, d\}$
e: $F = \{a, c, e\}$
f: $F = \{f\}$.

Note that the minimal and maximal solutions are not necessarily unique.

5.3 Solver implementation

We use a BDD-based solver (see Annex B for a short introduction to BDDs). Let S be the BDD that represents $[\![\mathcal{S}]\!]$.

5.3.1 Computation of the sets of enabled/disabled clocks

For any $c \in \mathcal{C}$, $\mathcal{F}^c = \emptyset$ means that c never appears in any solution v in its positive form (*i.e.,* where $v(c) = 1$). In terms of BDD this is expressed as "the cofactor of c in S is 0", where 0 is the void BDD. Remind that the cofactor of c in S (denoted $\mathsf{S_c}$) is obtained by substituting 1 for each occurrence of c in S.

$$(\forall c \in \mathcal{C}) c \in D \Leftrightarrow \mathsf{S_c} = 0 \tag{66}$$

Algorithm 1 computes the sets D and E:

Algorithm 1 Computation of E and D: ComputeSets(S)

Require: S is a BDD
Ensure: E is the set of enabled clocks and D the set of disabled clocks.
  $D \leftarrow \emptyset$
  $E \leftarrow \emptyset$
  for all $c \in \mathcal{C}$ **do**
    if $\mathsf{S_c} = 0$ **then**
      $D \leftarrow D \cup \{c\}$
    else
      $E \leftarrow E \cup \{c\}$
    end if
  end for
  return D,E

Example : No cofactor is 0, therefore D is empty and $E = \{a, b, c, d, e, f\}$.

5.3.2 Computation of required clocks

Given a clock c, as explained above, the cofactor $\mathsf{S_c}$ represents the set $\mathcal{F}^c$. c required c' at χ implies that for any solution v in $\mathcal{F}^c$, $v(c') = 1$, or equivalently, there is no v in $\mathcal{F}^c$ where $v(c') = 0$. In terms of BDD, this property can be stated as "$\mathsf{S_c} \wedge \neg c' = 0$". Hence, algorithm 2.

Algorithm 2 Computation of the required set of c at χ: ComputeReq(c, E, S)

Require: c is a clock, E the set of enabled clocks, S is the BDD
Ensure: The result is the required set of c at χ.

```
R ← {c}
for all c' ∈ E do
  if c ≠ c' then
    if S_c ∧ ¬c' = 0 then
      R ← R ∪ {c'}
    end if
  end if
end for
return R
```

Example : ComputeReq(e,E,S) returns $\{e, a, c\}$.

5.3.3 Computation of a random fireable subset of clocks

Choosing a random subset of fireable clocks is just taking a random solution in $\mathcal{F}$. This can be done by a traversal of the BDD S and making a random choice between the 0-edge and the 1-edge, when both are possible.

Algorithm RandomSat (Alg. 3) is an adaptation of the AnySat function provided by BDD packages. The terminal nodes of the ROBDD are written in bold fonts (**0** and **1**), while the bit values 0, 1 are written in normal font. Toss() is a function returning true or false equiprobably. The idea is to find a path from the top of the ROBDD to the terminal **1**.

The reader should note that it is impossible for $u.high$ and $u.low$ to be both equal to **0** in a (non **0**) ROBDD. This traversal algorithm is linear in the depth of the RBDD ($O(|\mathcal{C}|)$).

Example : Let $\times$ stand for the *don't care* bit value (0 or 1). Assume the variable ordering is $a < b < c < d < e < f$. RandomSat(S) sets v to one of the following solutions: (110$\times$0$\times$), (10101$\times$), (10000$\times$), (00000$\times$).

Algorithm ComputeRandom (Alg. 4) returns a fireable subset of $\mathcal{C}$. $\mathsf{S} = \mathbf{0}$ is the trivial case when no clocks are fireable.

This algorithm based on RandomSat(), which is linear in the depth of S, is far less expansive than Algorithm ComputeReq (Alg. 2). RandomSat(S) lets some variables free

Algorithm 3 Variant of the BDD's AnySat: RandomSat(u)

Require: u an internal node of a ROBDD and $u \neq \mathbf{0}$ { Recursive function modifying the global array v }
  if $u = \mathbf{1}$ **then**
    return {Terminal node **1** reached}
  else
    if $u.high = \mathbf{0}$ **then**
      $v(u.var) \leftarrow 0$
      RandomSat($u.low$)
    else
      if $u.low = \mathbf{0}$ **then**
        $v(u.var) \leftarrow 1$
        RandomSat($u.high$)
      else {both successors are possible}
        if Toss() **then**
          $v(u.var) \leftarrow 1$
          RandomSat($u.high$)
        else
          $v(u.var) \leftarrow 0$
          RandomSat($u.low$)
        end if
      end if
    end if
  end if

Algorithm 4 Computation of a random fireable set at χ: ComputeRandom(S)

Require: S the ROBDD
Ensure: The result is the random fireable set at χ.
  $R \leftarrow \emptyset$
  if S $\neq \mathbf{0}$ **then**
    $v \in \{0, 1, \times\}^n \leftarrow \times^n$ {$\times$ stands for the *don't care* truth value}
    RandomSat(S) {side-effect: modifies v}
    for all $c \in \mathcal{C}$ **do**
      Let $c' = \pi(c)$
      if $\big(v(c') = 1\big)$ or $\Big(\big(v(c') = \times\big)$ and Toss() $\Big)$ **then**
        $R \leftarrow R \cup \{c\}$
      end if
    end for
  end if
  return R

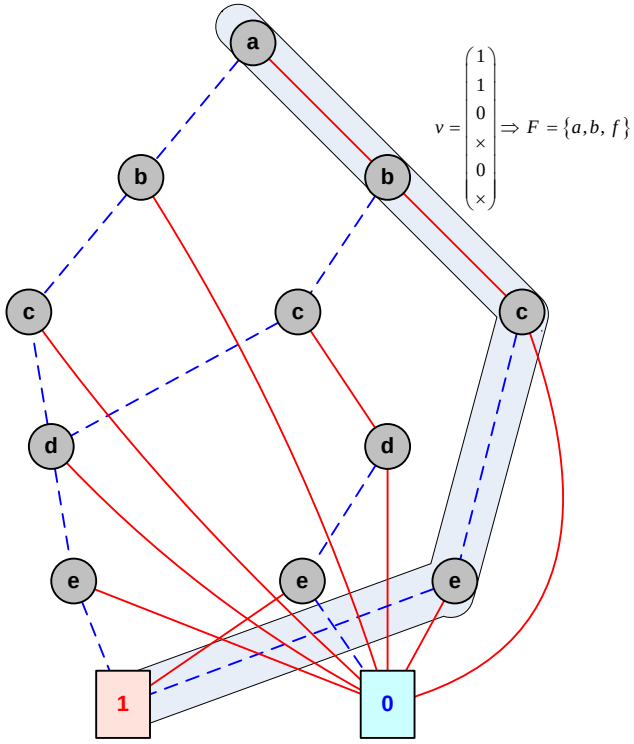

Figure 4: ROBDD and a possible traversal

(don't care value). These variables are then arbitrarily set to either 0 or 1. A variant taking the substitution by 0 systematically, would not be adequate because it could discard causal solutions. For instance, changing (110×0×), (10101×), (10000×), (00000×) respectively to (110000), (101010), (100000), (000000), never fires $\{a, b, d\}$, a solution which reflects the constraints $d \Rightarrow b$ and $b \Rightarrow a$. Even worse, f is never fired.

RandomSat is more general and less complex than ComputeReq, but it does not restrict solutions to causally dependent clock firings. It is up to the user to choose the most appropriate algorithm.

Example : Figure 4 is the ROBDD for the example. The variable ordering is $a < b < c < d < e < f$. Note that there is no node labeled f because f is not constrained in the specification. A traversal of the ROBDD has been shaded on the picture. For this path, RandomSat() has set v to (110×0×), and ComputeRandom() has chosen the firing set $F_1 = \{a, b, f\}$. $F_2 = \{a, b\}$, $F_3 = \{a, b, d\}$, $F_2 = \{a, b, d, f\}$ might have been chosen as well.

Note that the left-most path in Figure 4 yields $v = (00000\times)$, so that the firing set can be empty. This may be useful to model "stuttering". If this kind of behavior is unwanted, a solution consists in imposing that for each step, one clock at least fires. This can be done by adding a Boolean expression that is disjunction of all the Boolean variables: $\mathsf{S} \leftarrow \mathsf{S} \wedge \bigvee_{\mathsf{c} \in \mathscr{C}} \mathsf{c}$.

6 Conclusion

This report has described the syntax and the semantics of a kernel CCSL. The underlying formal models are the *Clock Model* and the *Time System*. The Clock Model $\mathcal{M}$ is a structure that consists of a finite set of discrete clocks $\mathcal{C}$ and a clock constraint specification $\mathcal{S}$ written in CCSL (or more precisely in a subset of CCSL: KCCL). The Time System consists of a Clock model and a current *configuration* χ, which indicates the index of the current instant of each clock. A Time System changes its state by *firing* clocks. A set of fired clocks is named a *step*. The dynamics of a Time model is represented by *runs*, which are sequences of steps. Of course, all firings must respect the clock constraints $\mathcal{S}$. A structural transformation of the KCCL specifications into a semantically equivalent Boolean expression has been given and illustrated on a simple example. This transformation is fully deterministic. Using a BDD-based Boolean solver, a set of fireable clocks is computed. Generally, this solution is not unique. Various policies are available to select one solution.

Clock constraints rely on three basic relations between pairs of instants: *precedence* $\prec$, *exclusion* $\#$, and *coincidence* $\equiv$. The first two are classical in models such as Petri nets. They express asynchronous dependency. The third relation departs from usual concurrency models and it corresponds to synchronous dependency, a concept borrowed from *synchronous reactive models*. A clock constraint imposes (infinitely) many instant relations. The parallel composition of clock constraints, interpreted as a conjunction of constraints, allows the specification of complex time constraints. This has been illustrated in several papers under different aspects:

- **Time model in the UML profile MARTE** . This model has been defined to support both "chronometric" time and *logical* time. CCSL has been introduced in Annex C3 of the MARTE specification. The paper entitled "Modeling Time(s)" [5], presented at MoDELS'07, describes these two forms of time and introduced the concept of clock constraints.

- **Modeling of time constraints** with application to the automotive domain. Logical time is suitable for modeling complex synchronization among events. This has been illustrated on a *knock detection and control* system [7]. Another paper analyzes the end-to-end timing constraints in an ABS system [8].

- **Relation with other formal models** A paper [9] presented at ISORC'09, compares the CCSL approach to (Time) Petri nets and the synchronous language Signal.

- **Improving the semantics of models**. Standards, like AADL, have a semantics generally given in natural languages. This may give rise to ambiguity or even worse to inconsistency. CCSL has been used as a pivot language to make the semantics of the AADL communications more formal [10, 11].

- **CCSL and formal verification**. CCSL specifications, which rely on a formal semantics, are amenable to formal analysis. This has been illustrated on the formal verification of an Esterel program [12] (LCTES'09).

Even though CCSL appears to be very expressive, the version presented in this report is a too crude version to be easily extensible by the users. An ongoing research focuses attention on a parameterized version, offering possibilities to define new clock expressions and clock constraints as reusable modules in libraries. Last but not least, CCSL is supported by a software environment called TimeSquare, which can be downloaded on the project WEB site (`http://www.inria.fr/sophia/aoste/dev/time_square`).

References

[1] OMG. *Unified Modeling Language, Superstructure*, November 2007. Version 2.1.2 formal/2007-11-02.

[2] The ProMARTE Consortium. *UML Profile for MARTE, beta 2*. Object Management Group, June 2008. OMG document number: ptc/08-06-08.

[3] C. André and F. Mallet. Modèle de temps de marte et ccsl. In J-C. Rault, editor, *Actes de NEPTUNE 2009*, volume 89. GL&IS, Juin 2009.

[4] C. Fernández. Non-sequential processes. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 95–115. Springer-Verlag, Sept. 1986.

[5] 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.

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

[7] M-A. Peraldi-Frati C. André, F. Mallet. A multiform approach to real-time system modeling: Application to an automotive system. Technical Report RR-2007-14, I3S, Sophia-Antipolis, (F), April 2007. `http://www.i3s.unice.fr/%7Emh/RR/2007/RR-07.14-C.ANDRE.pdf`.

[8] Frédéric Mallet, Marie-Agnès Peraldi-Frati, and Charles André. Marte CCSL and East-ADL2 timing requirements. Research Report 6781, INRIA, 12 2008.

[9] Frédéric Mallet and Charles André. UML/MARTE CCSL, signal and petri nets. Research Report 6545, INRIA, 05 2008.

[10] Charles André, Frédéric Mallet, and Robert de Simone. Modeling of immediate vs. delayed data communications: from AADL to UML marte. In *FDL*, pages 249–254. ECSI, September 2007.

[11] Frédéric Mallet, Robert de Simone, and Laurent Rioux. Event-triggered vs. time-triggered communications with UML Marte. In *FDL*, pages 154–159. IEEE, September 2008.

[12] Charles André and Frédéric Mallet. Combining CCSL and Esterel to specify and verify time requirements. Research Report 6839, INRIA, 02 2009. published in the proceedings of LCTES'09.

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:

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

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:

$$b^n = b \bullet b^{n-1} \quad (b \in \mathbb{B}, n \in \mathbb{N}^\star) \tag{67}$$

$$b^0 \triangleq \varepsilon \tag{68}$$

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:

$$\text{Let } b \in \mathbb{B}, u, v \in \mathbb{B}^*, u \bullet b\,(v \bullet b)^\omega = u\,(b \bullet v)^\omega \tag{69}$$

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.

$$w \downarrow k \triangleq |w[1..k]|_1$$

Let $k \in \mathbb{N}^\star, b \in \mathbb{B}, w$ a binary word

$$w \downarrow 0 \triangleq 0 \tag{70}$$
$$b \bullet w \downarrow k = b + (w \downarrow (k-1)) \tag{71}$$

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.

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

Let $k \in \mathbb{N}^\star, w$ a binary word

$$w \uparrow 0 \triangleq 0 \tag{72}$$
$$w \uparrow k \triangleq \omega \text{ if } |w|_1 < k \tag{73}$$
$$(1 \bullet w) \uparrow k = 1 + w \uparrow (k-1) \tag{74}$$
$$(0 \bullet w) \uparrow k = 1 + w \uparrow k \tag{75}$$

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:

$$(0 \bullet w_1) \circ w_2 = 0 \bullet (w_1 \circ w_2)$$
$$(1 \bullet w_1) \circ (b \bullet w_2) = b \bullet (w_1 \circ w_2) \text{ (for } b \in \mathbb{B})$$
$$(0 \bullet w_1) \circ \varepsilon = 0 \bullet (w_1 \circ \varepsilon)$$
$$(1 \bullet w_1) \circ \varepsilon = \varepsilon$$
$$\varepsilon \circ w_2 = \varepsilon$$

Properties:

$$|w_1 \circ w_2| = \min\{|w_1|, w_1 \uparrow (|w_2| + 1) - 1\} \tag{76}$$

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:

$$(b_1 \bullet w_1) + (b_2 \bullet w_2) = (b_1 \text{ or } b_2) \bullet (w_1 + w_2)$$
$$\varepsilon + w = w$$
$$w + \varepsilon = w$$
$$\varepsilon + \varepsilon = \varepsilon$$

B Binary Decision Diagram

Definition B.1 (BDD)**.** A *Binary Decision Diagram* (BDD) is a rooted, directed acyclic graph with

- one or more terminal nodes of out-degree 0 labeled 0 or 1, and
- a set of *internal nodes* u of out-degree 2. The successors of u are $u.low$ and $u.high$. The Boolean variable $u.var$ is associated with u.

Definition B.2 (OBDD)**.** An *Ordered BDD* (OBDD) is a BDD such that on all paths the variables respect a given linear order $x_1 < x_2 < \cdots < x_n$.

Definition B.3 (ROBDD)**.** A *Reduced OBDD* (ROBDD) is an OBDD such that

- (**uniqueness**) no two distinct nodes u and v have the same variable name and low- and high-successor, *i.e.,*

$$(u.var = v.var) \wedge (u.low = v.low) \wedge (u.high = v.high) \Rightarrow u = v$$

- (**non-redundant tests**) no internal node has identical low- and high-successor, *i.e.,*

$$u.low \neq u.high$$

A ROBDD can represent any Boolean function. Each node u defines a Boolean expression t^u such that

$$t^0 = 0$$
$$t^1 = 1$$
$$t^u = \text{if } u \text{ then } t^{u.high} \text{ else } t^{u.low}$$

Given a variable ordering $x_1 < x_2 < \cdots < x_n$, with each node u of the ROBDD, we associate the function f that maps $(b_1, b_2, \cdots, b_n) \in \mathbb{B}^n$ to the truth value of $t^u[b_1/x_1, b_2/x_2, \cdots, x_n/b_n]$.

Theorem B.1 (Canonicity of ROBDD)**.** *For any function $f : \mathbb{B}^n \to \mathbb{B}$ there is exactly one ROBDD u with variable ordering $x_1 < x_2 < \cdots < x_n$ such that $f^u = f(x_1, \cdots, x_n)$.*

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