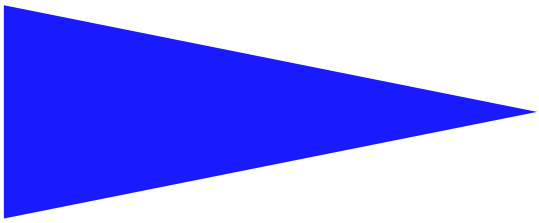
Compositional synthesis of latency-insensitive systems from multi-clocked synchronous specifications

Jean-Pierre Talpin, Dumitru Potop-Butucaru, Julien Ouy, Benoit Caillaud

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1730



COMPOSITIONAL SYNTHESIS OF LATENCY-INSENSITIVE SYSTEMS FROM MULTI-CLOCKED SYNCHRONOUS SPECIFICATIONS

JEAN-PIERRE TALPIN, DUMITRU POTOP-BUTUCARU, JULIEN OUY AND BENOîT CAILLAUD

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Compositional synthesis of latency-insensitive systems from multi-clocked synchronous specifications*

Jean-Pierre Talpin, Dumitru Potop-Butucaru, Julien Ouy and Benoît Caillaud

Systèmes communicants
Projet Espresso

Publication interne n ° 1730 — June 2005 — 22 pages

Abstract: We consider the problem of synthesizing correct-by-construction globally asynchronous, locally synchronous (GALS) implementations from modular synchronous specifications. This involves the synthesis of asynchronous wrappers that drive the synchronous clocks of the modules and perform input reading in such a fashion as to preserve, in a certain sense, the global properties of the system. Our approach is based on the weakly endochronous synchronous model, which gives criteria guaranteeing the existence of simple and efficient asynchronous wrappers. We focus on the transformation (by means of added signalling) of the synchronous modules of a multiclock synchronous specification into weakly endochronous modules, for which simple and efficient wrappers exist.

Key-words: Synchronous programming, separate compilation, compositionally, distribution

(Résumé : tsvp)

* This work is partly funded by the ARTIST2 Network of Excellence and the Regional Council of Brittany

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Compilation modulaire de systèmes insensibles à la latence à partir de spécifications synchrones multi-horloges

Résumé : Nous considérons le problème de transformer une spécification fonctionnelle synchrone multi-horloge en une mise en œuvre globallement asynchrone de manière modulaire et correcte par construction. Pour cela, nous élaborons une technique de compilation permettant de transformer une spécification déclarative synchrone en un automate effectuant des opérations atomiques de lecture, de calcul et d'écriture et assurant un invariant global d'insensibilité à la latence tout en préservant les propriétés locales de la spécification initiale.

Mots clés : Programmation synchrone, compilation séparée, compositionnalité, distribution

1 Introduction

Inspired by the concepts and practice of digital circuit design and automatic control, the *synchronous approach* has been used in the design of reactive real-time embedded software since the late '80s to facilitate the specification and analysis of control-dominated systems.

Provided that a few high-level constraints ensure compliance with the synchrony hypothesis, the designer can forget about timing and communication issues and concentrate on functionality. The synchronous model features deterministic concurrency and simple composition mechanisms facilitating the incremental development of large systems.

However, the problem of correctly implementing a synchronous specification does remain [2]. In particular, difficulties arise when the target implementation architecture has a distributed nature that does not match the synchronous assumption because of large variance in computation and communication speeds and because of the difficulty of maintaining a global notion of time. This is increasingly the case in complex microprocessors and Systems-on-a-Chip (SoC), and for many important classes of embedded applications in avionics, industrial plants, and the automotive industry.

Gathering advantages of both the synchronous and asynchronous approaches, the Globally Asynchronous Locally Synchronous (GALS) architectures are emerging as an architecture of choice for implementing complex specifications in both hardware and software. In a GALS system, locally-clocked synchronous components are connected through asynchronous communication lines. Thus, unlike for a purely asynchronous design, the existing synchronous tools can be used for most of the development process, while the implementation can exploit the more efficient/unconstrained/required asynchronous communication schemes.

Contributions In this paper, we consider the synthesis of correct and efficient GALS implementations for high-level, modular, synchronous specifications. This operation, also called *desynchronization* [1], involves the construction of asynchronous wrappers that satisfy 3 key properties:

- *predictability* – As the synchronous paradigm is often used in the development of safety-critical systems, input reading and the system itself must be deterministic, or at least predictable.
- *semantics preservation* – The GALS implementation must preserve the semantics of the synchronous specification (the set of asynchronous observations of synchronous traces must coincide with the set of traces of the GALS implementation).
- *efficiency* – The GALS implementation must minimize communication and component activation by taking into account as much as possible functioning modes and internal concurrency, and by allowing multi-rate computation.

Our approach is based on the micro-step atomata theory of [9], which gives high-level, implementation-independent conditions guaranteeing that simple asynchronous wrappers (*e.g.* wrappers that trigger a fireable reaction as soon as the needed input is available) produce correct and efficient GALS implementations. We therefore factor the synthesis problem into (1) a high-level, implementation-independent phase insuring the *weak endochrony* of the synchronous components and the *absence*

of deadlocks of the specification and (2) the actual wrapper synthesis phase, highly simplified by the high-level assumptions.

We focus on the analysis and the transformation of high-level modular synchronous and multi-clocked specifications (written in languages such as Signal, Lustre, or Esterel) to ensure the weak endochrony and absence of deadlocks. We define a new intermediate representation for synchronous programs, which does not suffer from the state explosion problem of the microstep automata of the weakly endochronous synchronous model (so that real-life systems can be represented and analyzed). At this level, we define symbolic analysis and synthesis algorithms that ensure the needed properties by means of added signalling.

Previous work Since the pioneering work of Caspi et al. [5], related approaches to implementing modular synchronous specifications on asynchronous architectures have explored many directions and models based on *Kahn process networks* (KPN), on *(generalized) latency-insensitive systems* (LIS), and on *(micro-step, weakly) endo-isochronous systems*. All approaches follow the same pattern as we do, trying to automatically distribute the components of a synchronous specification by the construction of wrappers ensuring global synchronization properties.

Related approaches have significant differences. The goal of the KPN-based approach [8] is the predictability (I/O determinism) of the implementation, but it does not offer means to reason about semantics preservation. Latency insensitive design [4] and its variants aim at dissociating computation and communication in the development of complex systems on a chip (by making the behavior of the system independent from the latency of the different communication lines). The proposed communication protocols effectively simulate a single-clock system, which is inefficient, but simplifies implementation.

More closely related to our work are the results based on *endo/isochronous systems* and their variants. The main goal here is to provide conditions guaranteeing correct desynchronization at the design abstraction level of the synchronous model. Differences between approaches regard their ability to represent different aspects of the system (execution modes, concurrency, causality, communication through read-write primitives) in order to support realistic and efficient implementations.

The model of Benveniste *et al.* [1] accounts for execution modes and inter-component concurrency, but it is defined in a non-causal framework making it difficult to represent intra-component concurrency and achieve compositionality. A variant of the notion of *endochrony* has a central place in the compilation of the synchronous language Signal [7] and further refined by the principles of finite-flow preservation in [10] for GALS architectures verification purposes.

We shall also mention the *quasi-synchronous* approach of Caspi in [6], the loosely time-triggered systems of Caspi *et al.* [3], the flow-invariance approach of [7], which guarantee synchronization and semantics preservation properties based on sampling constraints relating the clocks of the different components and communication lines of a distributed implementation.

The results presented in this paper are based on the recent results [9], presented in section 3 and 5, and provide the first actual implementation of these concepts. Finally, an important inspiration in our work comes from the large corpus of works concerning the development of asynchronous and GALS digital circuits.

Outline In the remainder, Section 2 gives an informal outline of the issues considered in this paper centered on the Signal formalism. Sections 3 and 5 present our framework to reason on synchrony, causality, and asynchronous implementation correctness. To exemplify the use of the model, Section 4 defines the first micro-step operational semantics of the Signal synchronous language.

The main contribution of the paper is presented in Sections 6-8. Section 6 introduces a new intermediate representation for synchronous programs that is used to check weak endochrony (of modules) and deadlock-freedom (of architectures), Section 7, and to synthesis latency insensitive code, Sections 8.

2 Position of the problem

We position the problem by considering multi-clocked synchronous (or polychronous) specifications declared in the data-flow formalism Signal. A Signal process consists of the simultaneous composition of equations on signals that partially relate them with respect to an abstract timing model.

Polychrony In Signal [7], a process p is an infinite loop that consists of the synchronous composition $p|q$ of simultaneous equations $x = y\, f\, z$ over signals noted x, y, z. Restricting the lexical scope of a signal name x to a process p is noted p/x.

$$p, q \ ::= \ (x = y\, f\, z) \mid p|q \mid p/x$$

A network of synchronous processes is noted P and $P \parallel Q$ stands for the asynchronous composition of P and Q.

$$P, Q \ ::= \ p \mid P \parallel Q$$

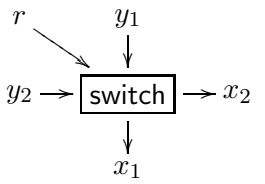
Equations Equations $x = y\, f\, z$ define partially-ordered timing relations between input and output signals. There are three primitive operators in Signal: delay, sampling and merge. A delay equation $x = y \,\mathsf{pre}\, v$ initially defines the signal x by the value v and then by the previous value of the signal y. In a delay equation, the signals x and y are assumed to be synchronous, i.e., either simultaneously present or simultaneously absent at all times.

A sampling $x = y \,\mathsf{when}\, z$ defines x by y when z is true and both y and z are present. In a sampling equation, the output signal x is present iff both input signals y and z are present and z holds the value true. A merge $x = y \,\mathsf{default}\, z$ defines x by y when y is present and by z otherwise. In a merge equation, the output signal is present iff either of the input signals y or z is present.

We observe that signals defined by the sampling and merge equations are partially synchronized: they do not declare any synchronization relation between their input signals y and z.

Structure The structuring element of a Signal specification is a process. A process accepts input signals, possibly from different clock domains, and produces output signals when needed. A process is hierarchically decomposed into sub-processes to structurally describe the functionalities of a system.

An example As a example, we consider a simple crossbar switch. Its interface is composed of two input data signals y_1 and y_2 and a reset input signal r. Data signals are routed along the output data signals x_1 and x_2 upon the internal state s of the switch. The state is toggled using the reset signal by the functionality $\mathsf{toggle}(s, r)$. Data is routed along an output signal x from two possible input sources y_1 or y_2 upon depending on the value of s by two instance of the functionality $x = \mathsf{route}(s, y_1, y_2)$.



$$(x_1, x_2) = \mathsf{switch}(y_1, y_2, r) \stackrel{\text{def}}{=} \left(\begin{array}{|ll} & \mathsf{toggle}(s, r) \\ x_1 = & \mathsf{route}(s, y_1, y_2) \\ x_2 = & \mathsf{route}(s, y_2, y_1) \end{array} \right) / s$$

The subprocess toggle defines the signal s that reflects the current state of the switch. It reads the previous value v of s with $s\,\mathsf{pre}\,\mathsf{true}$. If r is present and true, then it sends $\mathsf{not}\,v$ along s and else v (if r is either absent or false).

$$\mathsf{toggle}(s, r) \stackrel{\text{def}}{=} (s = (\,\mathsf{not}\,(s\,\mathsf{pre}\,\mathsf{true}\,)\,\mathsf{when}\,r)\,\mathsf{default}\,(s\,\mathsf{pre}\,\mathsf{true}\,))$$

The subprocess route selects which of the values v and w of its input signals y or z to send along its output signal x depending on the boolean signal s. If s is present and true, it chooses v and else, if s is present and false, it chooses w.

$$x = \mathsf{route}(s, y, z) \stackrel{\text{def}}{=} (x = (y\,\mathsf{when}\,s)\,\mathsf{default}\,(z\,\mathsf{when}\,\mathsf{not}\,s))$$

Remember that data-flow equations in Signal partially synchronize input and output signals. In the route process, this implies that none of the signals y, z and s are synchronized, and that the output signal x is present iff either y is present and s true or z is present and s false.

The issue We easily observe that the switch is sensitive to latency by composing it with the following test sequence alternating y_2 and y_1 synchronized to r. The synchronous composition of the switch with the test process can only produce one possible trace (up to stuttering) by relying to the implicit synchronization relation linking the signals r, $y_{1,2}$ and $x_{1,2}$.

$$\begin{array}{rcccc} \mathsf{test} : & t_1 & t_2 & t_3 & t_4 \\ y_1 : & 1 & & 1 & \\ y_2 : & & 1 & & 1 \\ r : & 1 & & 1 & \end{array} \qquad \begin{array}{rcccc} \mathsf{test}\,|\,\mathsf{switch} : & t_1 & t_2 & t_3 & t_4 \\ x_1 : & & 1 & & 1 \\ x_2 : & 1 & & 1 & \end{array}$$

By contrast, the asynchronous composition $\mathsf{test} \parallel \mathsf{switch}$ has many possible (and non stuttering-equivalent) traces in the absence of these relation between the data input signals $y_{1,2}$ and the reset signal r.

$$\begin{array}{rcccc} \mathsf{test} \parallel \mathsf{switch} : & t_1 & t_2 & t_3 & t_4 \\ x_1 : & & 1 & & \\ x_2 : & 1 & 1 & & 1 \end{array} \quad \ldots \quad \begin{array}{rcccc} & t_1 & t_2 & t_3 & t_4 \\ x_1 : & & 1 & & 1 \\ x_2 : & 1 & & 1 & \end{array} \ldots$$

Desynchronization The topics of desynchronization addresses this very issue. It consists of interfacing the processes switch and test with automatically synthesized protocols ensuring a flow of information that is not sensitive to communication latency. In the related work, this is achieved by imposing local timing constraints and global synchronizations.

- Local timing constraints are imposed to individual processes to provide a guarantee of *endochrony*. Endochrony is the property of a process that is capable of reconstructing the the presence/absence status of all its signals starting from a set of asynchronous input streams.

  In the case of the switch, a way to impose endochrony consists of adding two input signals $z_{1,2}$, synchronized to r and s and that hold the value true iff $y_{1,2}$ are present. The syntax $x \,\mathsf{sync}\, y$ defines a synchronization relation $\hat{x} = \hat{y}$ between the clocks $\hat{x}$ and $\hat{y}$ of x and y. An unary $\mathsf{when}\, x$ stands for the clock x.

$$\mathsf{wrap}(r, s, y_i, z_i) \stackrel{\text{def}}{=} ((z_i \,\mathsf{sync}\, r \,\mathsf{sync}\, s) \,|\, (y_i \,\mathsf{sync}\, \mathsf{when}\, z_i))$$

- Global synchronizations are imposed by an ad-hoc protocol to exchange clocks between individual processes so as to provide a guarantee of *isochrony* (isochronous processes communicate endochronously by exchanging a master communication clock). In the case of the switch and test processes, isochrony is achieved by pacing all communications to the signal z.

The main drawback of this technique is that it is not compositional: the interfaces and protocols specifically built for the switch and the test functions need to be regenerated every time a functionality is added to the system.

Our approach The aim of the present article is to look at the issue of desynchronization in a radically different way by considering the theory of weakly-endochronous micro-step automata [9]. This theory provides a compositional way to locally weaken the constraints under which a process is guaranteed to be robust to communication latency. Consequently, clock exchange rates are globally reduced. The proposed technique focuses on the synthesis of weakly endochronous modules as its applications largely escape the specific topics of desynchronization and allow for the separate compilation and static or dynamic scheduling of synchronous processing units.

3 Model of micro-step automata

We start our presentation with a summary of the theory under consideration. The micro-step automata theory under consideration is a particular class of concurrent automata equipped with synchronous product and synchronous and asynchronous FIFO models allowing to represent computation and communication causality as well as communication through read/write primitives over given communication channels. A detailed description of this theory can be found in [9].

Micro-step automata *Micro-step automata* communicate through signals $x \in X$. The *labels* $l \in L_X$ generated by the set of names X are represented by a partial map of *domain* from a set of

signals X noted $\text{vars}(l)$ to a set of values $V^\perp = V \cup \{\perp\}$ and tags. The label $\perp$ denotes the *absence* of communication during a transition of the automaton.

We note $l' \leq l$ iff there exists l'' disjoint from l' such that $l = l' \cup l''$ and then $l \setminus l' = l''$. We say that l and l' (resp. t and t') are *compatible*, written $l \bowtie l'$, iff $l(x) = l'(x)$ for all $x \in \text{vars}(l) \cap \text{vars}(l')$ and, if so, note $l \cup l'$ their union. We write $\text{supp}(l) = \{x \in X \mid l(x) \neq \perp\}$ for the *support* of a label l and $\perp_X$ for the empty support.

Definition 1 *An* automaton $A = (s^0, S, X, \rightarrow)$ *is defined by an initial state* s^0*, a finite set of states* S *noted* s *or* $x = v$*, labels* L_X *and by a transition relation* $\rightarrow$ *on* $S \times L_X \times S$*.*

The product $A_1 \otimes A_2$ *of* $A_i = (s_i^0, S_i, X_i, \rightarrow_i)$ *for* $0 < i \leq 2$ *is defined by* $((s_1^0, s_2^0), S_1 \times S_2, X_1 \cup X_2, \rightarrow)$ *where* $(s_1, s_2) \rightarrow^l (s_1', s_2')$ *iff* $s_i \rightarrow^{l|_{X_i}} s_i'$ *for* $0 < i \leq 2$ *and* $l|_{X_i}$ *the projection of* l *on* X_i*.*

An automaton $A = (s^0, S, X, \rightarrow)$ *is* concurrent *iff* $s \rightarrow^\perp s$ *for all* $s \in S$ *and if* $s \rightarrow^l s'$ *and* $l' \leq l$ *then there exists* $s'' \in S$ *such that* $s \rightarrow^{l'} s''$ *and* $s'' \rightarrow^{l \setminus l'} s'$*.*

Synchronous automata *Synchronous automata* account for primitive communications using read and write operations on*directed communication channels* pairing variables x with directions represented by tags.

Emitting a value v along a channel x is written $!x = v$ and receiving it $?x = v$. We write $\text{vars}(D)$ for the channel names associated to a set of directed channels D. The undirected or untagged variables of a synchronous automaton are its *clocks* noted c.

Definition 2 *A* synchronous *automaton* $(s^0, S, X, c, \rightarrow)$ *of clock* $c \in X$ *is a concurrent automaton* $(s^0, S, X, \rightarrow)$ *s.t.*

1. $s \rightarrow^l s$ *implies* $l = c$ *or* $c \not\leq l$
2. $s^0 \rightarrow^c s^0$
3. $s \rightarrow^c s'$ *implies* $s' \rightarrow^c s'$
4. *if* $s_{i-1} \rightarrow^{l_i} s_i$ *and* $l_i \neq 1$ *for* $0 < i, j \leq n$ *then* $\text{vars}(l_i) \cap \text{vars}(l_j) = \emptyset$ *iff* $i \neq j$*.*

We assume that a channel x connects at most one emitter with at most one receiver. Multicast will however be used in examples and is modeled by substituting variable names (one $!x = v$ and two $?x = w_{1,2}$ will be substituted by two $!x = v$, $!x_2 = v$ and two $?x = w_1$ $?x_2 = w_2$ by introducing a local signal x_2).

For an automaton A, we define a *trace* $t \in T = L_X^*$ by a finite sequence of labels, write $|t|$ for its length and refer to t_i as the ist label in t and $\mathcal{T}_A(s)$ as the set of traces accessible by A from state s.

For a synchronous automaton of clock c, we write $s \rightarrow_t^* s'$ iff there exists a series $(s_i)_{0 \leq i \leq n}$ with $n = |t|$ such that $s_0 = s$, $s_{i-1} \rightarrow^{t_i} s_i$ for $0 < i \leq n$ and $s_n = s'$. We note $s_0 \twoheadrightarrow^t s$ iff $s \rightarrow_t^* s'$ with $t_i \neq c$ for $0 < i \leq |t|$ and $s_{|t|} \rightarrow^c s$.

Composition of automata *The composition of automata* is defined by synchronized product, using first-in-first-out buffer models to represent communication through synchronous and asynchronous channels.

Synchronous communication is modeled using 1-place synchronous FIFO buffers. The synchronous FIFO of clock c and channel x is noted sFIFO_c^x and the asynchronous FIFO buffer of channel x is written aFIFO^x.

A synchronous FIFO buffer serializes the emission event $!x = v$ followed by the receipt event $?x = v$ within the same transition (the clock tick c occurs after). Silent transitions $s_i \rightarrow^{\perp} s_i$ for $0 \leq i \leq 2$ are left implicit as sFIFO_c^x is assumed to be a concurrent automaton.

$$\text{sFIFO}_c^x \stackrel{\text{def}}{=} \left(s_0, \{s_{0..2}\}, \{?x, !x, c\}, c,\ c \circlearrowright s_0 \xrightarrow{!x=v} s_1 \xrightarrow{?x=v} s_2 \xrightarrow{c} s_0 \right)$$

The model of an unbounded and asynchronous FIFO buffer is similarly defined by the repetition of the communication pattern of the synchronous buffer and by induction on its storage size represented by a word $w \in V^*$.

$$\text{aFIFO}^x \stackrel{\text{def}}{=} \left(\begin{array}{l} \epsilon, V^*, \cup_{v \in V} \{?x = v, !x = v\}, \\ \quad \cup_{n \geq 0} \left\{ vw \xrightarrow{?x=v} w \xrightarrow{!x=v} wv \,|\, v \in V,\, w \in V^n \right\} \end{array} \right)$$

Two synchronous automata are *composable* if their tagged variables are mutually disjoint. This rule enforces the point-to-point communication restriction previously mentioned and non-overlapping of clocks.

The synchronous composition of two automata consists of its synchronous product with the synchronous FIFO buffer model instantiated for all channels x common to the vocabulary of the composed automata. The clocks $c_{1,2}$ of both automata are synchronized to the clock c of the FIFO (by the substitution $A_i[c/c_i]$ of c_i by c in A_i).

Similarly, the asynchronous composition consists of the synchronous product of the composed automata with instances of asynchronous FIFO buffers for all common channels x and, this time, without clock synchronization.

Definition 3 *Let $A_i = (s_i^0, S_i, X_i, c_i, \rightarrow_i)_{i=1,2}$ be two composable synchronous automata and c a clock and write $A[c_2/c_1]$ for the substitution of c_1 by c_2 in A.*

If $A = (s, S, X, c, T)$ then the restriction A/x is defined by $(s, S, X/\{x\}, c, \{s_1 \rightarrow^{l/x} s_2 \in T' \mid s_1 \rightarrow^{l} s_2 \in T\}$.

The synchronous composition $A_1 \,|^c A_2$ is defined by the product of A_1, of A_2 and of a series of synchronous FIFO buffers sFIFO_c^x that are all synchronized at the same clock c.

$$A_1 \,|^c A_2 = (A_1[c/c_1]) \otimes \left(\bigotimes_{x \in \text{vars}(X_1)}^{x \in \text{vars}(X_2)} \text{sFIFO}_c^x \right) \otimes (A_2[c/c_2])$$

The asynchronous composition $A_1 \parallel A_2$ is defined by the product of A_1 and A_2 with asynchronous FIFO buffers aFIFO^x for all $x \in \text{vars}(X_1) \cap \text{vars}(X_2)$.

$$A_1 \,\|^c A_2 = A_1 \otimes \left(\bigotimes_{x \in \text{vars}(X_1)}^{x \in \text{vars}(X_2)} \text{aFIFO}^x \right) \otimes A_2$$

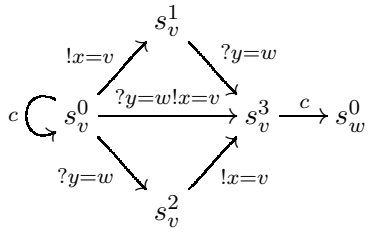
4 Micro-step semantics of Signal

Micro-step automata provide an expressive operational semantics framework for the multi-clocked specification formalism Signal under consideration. We therefore detail the automata of some operators presented in section 2.

- *A delay equation* $p =^{\text{def}} (x = y \,\mathsf{pre}\, v_0)$ corresponds to an automata composed of four micro-states $s_v^{0..3}$ and for each value v of the signal x and starting from an initial state $s_{v_0}^0$ with the initial value v_0.

  The automaton concurrently performs both receive $?y = w$ and send $!x = v$ actions and then issues a clock transition c to the state s_w^0 where the next reaction takes place.

$$A_{(x=y \,\mathsf{pre}\, v_0)} = \left(s_{v_0}^0, \{s_v^{0..3}, \mid v \in V\}, \{x, y\}, c, \bigcup_{v,w \in V} \left(\begin{array}{c} \\ \\ \end{array} \right. \right.$$



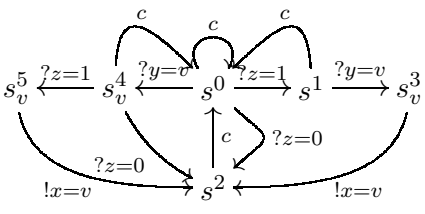
$$\left. \left. \begin{array}{c} \\ \\ \end{array} \right) \right)$$

- *A sampling equation* $p =^{\text{def}} (x = y \,\mathsf{when}\, z)$ starts from its initial state s^0 and concurrently performs either of two receive actions $?y = v$ and $?z = w$. If either y or z is absent or if z equals 0 then the automaton performs a clock transition to the initial state s^0.

  Otherwise, it performs the send action $!x = v$. This means that x is causal to both y and z. Notice that all intermediate states are parameterized with the value v under consideration.

$$A_{(x=y \,\mathsf{when}\, z)} = \left(s^0, \{s^{0..2}, s_v^{3..5} \mid v \in V\}, \{x, y, z\}, c, \right.$$

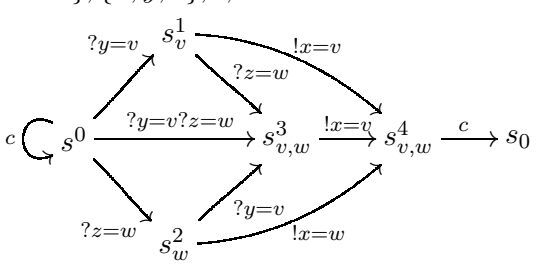
$$\left. \bigcup_{v \in V} \left(\begin{array}{c} \\ \\ \end{array} \right) \right)$$



- *A merge equation* $p =^{\text{def}} (x = y \,\mathsf{default}\, z)$ performs two concurrent receive actions $?y = v$ and $?z = w$. If y is present then the send action is always $!x = v$ to s_4. If only z is present (in s^2) then the send action is $!x = w$ to s^4 before a clock transition back to s^0.

Otherwise the only choice is a silent transition at s_0. Again, all intermediate states are parameterized with the possible pairs $(v, w) \in V^2$.

$$A_{(x=y \text{ default } z)} = \left(s^0, \{s^1_v, s^2_w, s^{3..4}_{v,w} \mid v, w \in V\}, \{x, y, z\}, c, \bigcup_{v,w \in V} \right.$$



$$\left. \right)$$

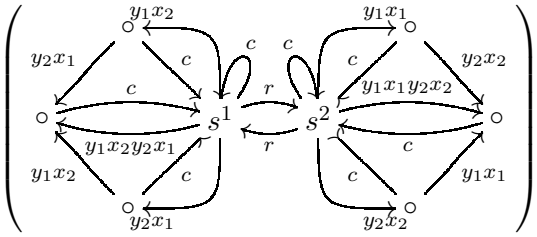
- *Composition $p \mid q$ or $P \parallel Q$ and restriction p/x* are defined by structural induction starting from the previous axioms and by using equivalent composition concepts of the model of micro-step automata.

$$A_{p \mid q} = A_p \mid^c A_q \qquad A_{p/x} = (A_p)/x \qquad A_{P \parallel Q} = A_P \parallel A_Q$$

Example 1 *Despite the fact that micro-state automata expose many intermediate states to adequately model causality, their synchronous composition does not necessarily yield to an explosion of the state space (only macro-step transitions $\twoheadrightarrow$ do). As an example, consider the transition system for the* switch *process (the notation $y_j x_i$ stands for two micro-transitions $\circ \rightarrow^{?y_j=v} \circ \rightarrow^{!x_j=v} \circ$).*

The switch automaton consists of two, mirrored, structures that allow for concurrently receiving y_1 and y_2 and transmitting them along x_1 or x_2 according to the mode s_1 or s_2, toggled using the signal r.

$$T_{\text{switch}} =$$



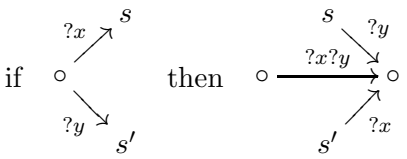
We easily observe the possibility of the switch to non-deterministically choose to either perform a routing y_1x_1 or a toggle r from s^0 from desynchronized inputs y_1 and r.

5 Formal properties

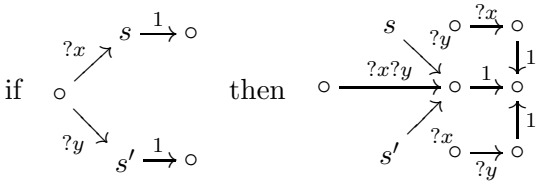
To address this issue, the property of weak endochrony [9] defines the class of deterministic micro-automata which satisfy insensitivity to latency. Informally, weak endochrony is meet by a deterministic automaton if three conditions are satisfied:

- Transitions sharing no variable are independent, e.g., suppose an automaton with two transitions to s with the label $?x$ and to s' with the label $?y$. Since x and y are distinct signals,

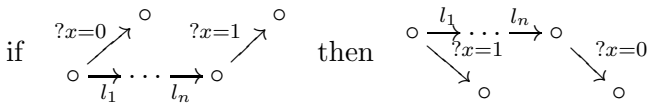
the automaton should have the ability to perform both $?x$ and $?y$ actions simultaneously, or $?y$ from s, or $?x$ from s', and arrive into the same final state.



- Non contradictory transitions can be united, e.g., suppose the same actions as above, but now followed by a clock transition 1 from s and s'. Then, similarly, the automaton should have the ability to perform $?y$ during the transition following s and $?x$ from the transition following s' or both, simultaneously.



- Choice does not change over time, e.g., suppose an automaton with a transition of label $?x = 0$ and a series of transitions $l_{1..n}$ (not mentioning x) to the alternative label $?x = 1$. Then the automaton should equally be able to choose $?x = 1$ first and $?x = 0$ after $l_{1..n}$.



Formally,

Definition 4 *Let us write $s \twoheadrightarrow^t$ iff there exists s' such that $s \twoheadrightarrow^t s'$. A micro-automaton $A = (s^0, S, X, c, \rightarrow)$ is* weakly-endochronous *iff it is synchronous and:*

1. deterministic*: if $s \rightarrow^l s_1$ and $s \rightarrow^l s_2$ then $s_1 = s_2$*
2. step-independent*: if $s \rightarrow^{l_1} s_1$, $s \rightarrow^{l_2} s_2$ and $\mathrm{supp}(l_1) \cap \mathrm{supp}(l_2) = \emptyset$ then there exists s' such that $s_1 \rightarrow^{l_2} s'$, $s_2 \rightarrow^{l_1} s'$ and $s \rightarrow^{l_1 \cup l_2} s'$*
3. clock-independent*: $s \rightarrow^c s'$ implies*
   (a) *if $s \xrightarrow{t}{}^*$ and $c \notin \mathrm{supp}(t)$ then $s' \xrightarrow{t}{}^*$*
   (b) *if $s \twoheadrightarrow^t s''$ then $s' \twoheadrightarrow^t s''$*
   (c) *if $s' \twoheadrightarrow^{tt''}$ then $s \twoheadrightarrow^{tt'}$ and $t' \leq t''$*
   (d) *if $s \twoheadrightarrow^t$ and $s \twoheadrightarrow^{t'}$ and $t \bowtie t'$ then $s \twoheadrightarrow^{t(t' \setminus t)}$*
4. choice-independent*: if $\mathrm{supp}(l_1) = \mathrm{supp}(l_2)$, $s \xrightarrow{t_1 l_1}{}^*$, $s \xrightarrow{t_2 l_2}{}^*$ and $t_1 \bowtie t_2$ then $s \xrightarrow{t_1 l_2}{}^*$ and $s \xrightarrow{t_1 l_2}{}^*$*

Example 2 *The micro-step automaton of the* switch *happens to be (micro-step) deterministic but it does not meet the criterion of step and clock independence in Definition 4. One has that:*

2. $s_0 \rightarrow^r s_1$, $s_0 \rightarrow^{y_1 x_1} \circ$ *but not* $\circ \rightarrow^r s'$

3d. $s_0 \rightarrow^c s_0$, $s_0 \twoheadrightarrow^{y_1 x_1}$ *and* $s_0 \twoheadrightarrow^r$ *but not* $s_0 \twoheadrightarrow^{y_1 x_1 r}$

Process switch *must be refined in a way to meet these properties. This will be the purpose of the sections 6 and 7 on the analysis and validation of multi-clocked specifications.*

We now state the property of weak isochrony by considering a series $A_i = (s_i^0, S_i, X_i, c_i, T_i), 0 < i \leq n$ of composable synchronous automata and note $B =^{\text{def}} \|^c_{0<i\leq n} A_i$ and $C =^{\text{def}} \|_{0<i\leq n} A_i$ their synchronous and asynchronous compositions. Let $\mathcal{T}_A(s)$ for the set of traces of an automaton A starting from its state s,

- The *injective morphism* $\mathcal{I}$ from $\mathcal{T}_B(s)$ to $\mathcal{T}_C(\mathcal{I}s)$ by induction on traces with $\mathcal{I}(\epsilon) = \epsilon$, $\mathcal{I}(tt') = \mathcal{I}(t)\mathcal{I}(t')$, $\mathcal{I}(c) = c_{1..n}$ and $\mathcal{I}(l) = l$ iff $l \neq c$.
- The *desynchronization morphism* $\mathcal{D}$ from $\mathcal{T}_B(s)$ to $\mathcal{T}_C\ (\mathcal{I}s)$ by induction on traces with $\mathcal{D}(\epsilon) = \epsilon$, $\mathcal{D}(tt') = \mathcal{D}(t)\mathcal{D}(t')$, $\mathcal{D}(\perp) = \epsilon$ and $\mathcal{D}(l) = l$ iff $l \neq \perp$.

Next, let $t \sqsubseteq t'$ iff $\mathcal{D}(t)|_x$ is a prefix of $\mathcal{D}(t')|_x$ for all $x \in \text{vars}(t)$ and asynchronous equivalence $t \sim t'$ iff $t \leq t'$ and $t' \leq t$. We say that C is a correct desynchronization of the synchronous specification B iff, for all state s of C, and all traces $t \in \mathcal{T}_B(\mathcal{I}s)$ there exists $t_1 \in \mathcal{T}_C(\mathcal{I}s)$ and $t_2 \in \mathcal{T}_B(s)$ such that $t \sqsubseteq t_1 \sim \mathcal{I}t_2$

Definition 5 *Let* $(A_i)_{0<i\leq n}$ *be weakly endochronous automata, if the synchronous composition* $\|^c A_i, 0 < i \leq n$ *is non-blocking (i.e.* $s \rightarrow^l s' \Rightarrow s \twoheadrightarrow^{lt}$*) then the* $(A_i)_{0<i\leq n}$ *are weakly isochronous.*

Weakly isochronous automata satisfy the desynchronization correctness criterion of [9].

Theorem 1 *If the automata* $A_i, 0 < i \leq n$ *are weakly isochronous then their desynchronization is correct.*

6 Flow analysis

In order to define decision criteria, section 7, to validate the properties of definitions 4 and 5 and meet the property of theorem 1, we give an intermediate representation of multi-clocked specification that exposes its control and data-flow properties for the purpose of their analysis.

A control and data-flow graph A process p is represented as a data-flow graph G. In this graph, a vertex g is a data-flow relation that partially defined a clock or a signal. A signal vertex $c \Rightarrow x = f(y_{1..n})$ partially defines x by $f(y_{1..n})$ at the clock c. We note $\hat{g}$ the clock c of a signal vertex g, use(g) its set of used signal names $\{y_{1..n}\}$, def(g) its defined signal name x, vars(g) = use(g) $\cup$ def(g) and fun(g) its function f, which can either be the identity, a boolean function ($\wedge, \vee$ or $\neg$) or the delay operator pre.

$$G, H ::= g \mid (G \| H) \mid G/x \qquad g, h ::= \hat{x} = e \mid c \Rightarrow x = f(y_{1..n})$$

Clocks A clock vertex $\hat{x} = e$ defines a relation between two clocks. A clock c defines a condition upon which a data-flow relation can be executed. It expresses control. The clock $\hat{x}$ defines when the signal x is present (its value is available). The clocks x and $\neg x$ mean that x is true and false, respectively, and hence present. A clock expression e is Boolean expression that defines the way a clock is computed. 0 means never.

$$c ::= \hat{x} \,|\, x \,|\, \neg x \qquad e ::= 0 \,|\, c \,|\, e_1 \vee e_2 \,|\, e_1 \wedge e_2$$

The decomposition of a process into the synchronous composition of clock and signal vertices is defined by induction on the structure of p. Each equation is decomposed into data-flow functions guarded by a condition, the clock $\hat{x}$ of the output. This clock will need to be computed for the function to be executed.

Notice the particular decomposition of a merge equation. The partial equations $x = y$ and $x = z$ are differentiated by the true and false value of a boolean signal δ, which we call a *differential clock*. A subtlety is that the sub-clock $\neg\delta$ is not locally defined while $\hat{\delta}$ and δ are. It is non-constructively defined by the difference between $\hat{z}$ and $\hat{y}$.

$$\begin{aligned}
G_{[x=y \,\mathsf{pre}\, v]} &= (\hat{x} \Rightarrow x = \mathsf{pre}\,(y, v)) \,|\, (\hat{x} = \hat{y}) \\
G_{[x=y \,\mathsf{when}\, z]} &= (\hat{x} \Rightarrow x = y) \qquad\qquad |\, (\hat{x} = \hat{y} \wedge z) \\
G_{[x=y \,\mathsf{default}\, z]} &= (\ (\delta \Rightarrow x = y) \qquad\quad |\, (\hat{\delta} = \hat{x}) \\
&\qquad\quad |\, (\neg\delta \Rightarrow x = z) \qquad |\, (\delta = \hat{y}) \\
&\qquad\qquad\qquad\qquad\quad |\, (\hat{x} = \hat{y} \vee \hat{z}))/\delta \\
G_{[p \,|\, q]} &= G_{[p]} \,|\, G_{[q]} \\
G_{[p/x]} &= G_{[p]}/x
\end{aligned}$$

Separation of concerns For the sake of a clear separation of concerns, the form of the graph G_p of a process p is decomposed into its clock vertices C_p, that defines its control and timing model, and signal vertices S_p, that define its (untimed) data-flow.

The set of bound signal names X_p of G_p is further extracted by commutativity and for any substitution $(G/x) \,|\, H = ((G[y/x]) \,|\, H)/y$ of x by $y \notin \mathrm{vars}(G) \cup \mathrm{vars}(H)$ in G to yield the decomposition of the graph G_p of a process p as

$$G_p =^{\mathrm{def}} (C_p \,|\, S_p)/X_p$$

Example 3 *Let us construct the graph of the crossbar switch. It can modularly be defined by one instance of the toggle and two instances of the router.*

$$G_{\mathsf{switch}} \stackrel{\mathrm{def}}{=} ((G_{\mathsf{toggle}}/st\delta) \mid (G_{\mathsf{route}_1}/\delta_1) \mid (G_{\mathsf{route}_2}/\delta_2))$$

Each functionality of the switch can then be further decomposed into its untimed data-flow graph and its specific timing model expressed by clock relations.

$$\begin{aligned}
S_{\mathsf{toggle}} &\stackrel{\mathrm{def}}{=} (\hat{s} \Rightarrow t = s \,\mathsf{pre}\, \mathsf{true}\,) \\
&\quad\ |\, (\delta \Rightarrow s = \mathsf{not}\, t) \\
&\quad\ |\, (\neg\delta \Rightarrow s = t)
\end{aligned}
\qquad
\begin{aligned}
C_{\mathsf{toggle}} &\stackrel{\mathrm{def}}{=} (\hat{s} = \hat{r}) \\
&\quad\ |\ (\hat{\delta} = \hat{s}) \\
&\quad\ |\ (\delta = \hat{s} \wedge r)
\end{aligned}$$

For all $0 < i \neq j \leq 2$, we note $e_i^1 = \hat{y}_i \wedge s$ and $e_j^2 = \hat{y}_j \wedge \neg s$ in the definition of the router.

$$S_{\mathsf{route}_i} \stackrel{\text{def}}{=} (\delta_i \Rightarrow x_i = y_i) \;\big|\; (\neg\delta_i \Rightarrow x_i = y_j) \qquad C_{\mathsf{route}_i} \stackrel{\text{def}}{=} (\hat{x}_i = \hat{\delta}_i) \,|\, (\hat{\delta}_i = e_i^1 \vee e_j^2) \;\big|\; (\delta_i = e_i)$$

Notations The remainder requires a couple of notations to be defined: we write $G \models e = f$ iff the system of Boolean equations C_p in G implies that $e = f$ always holds.

In addition, and for all process p and all boolean signal x in p, we assume that $C_p \models \hat{x} = x \vee \neg x$ and $C_p \models x \wedge \neg x = 0$. We note $c \leq d$ for syntactic clock inclusion: $x < \hat{x}$ and $\neg x < \hat{x}$ and $\hat{x} \leq \hat{x}$.

We write $\mathrm{vars}(p)$ for the set of free signal names of a process p. The free signals of a process are those which appear in equations and whose scope is not bound by restriction. The free output signals $\mathrm{out}(p)$ of a process p are free signal names occurring on the left-hand side of equations.

The free input signals $\mathrm{in}(p)$ of a process p are the remainder $\mathrm{vars}(p) \setminus \mathrm{out}(p)$. The state variables $\mathrm{state}(p)$ of a process p are bound signals $x \in X_p$ defined by a delay equation.

Control-flow analysis The flow analysis of graphs G_p comprises control-flow aspects whose aim is to determine signal clocks from the information available to the process p: its input signals interface and its clock relations C_p.

To this end, the relation $c \prec C$ is defined between a clock c and the set of supposedly known clocks C is can be deduced from in order to express it by a Boolean function f that satisfies $C_p \models c = f(C)$.

Definition 6 *The clock c is computable from the input signals $\mathrm{in}(p)$, written $c \prec C$, iff*
- *if $x \in \mathrm{in}(p)$ and $c \leq x$ then $c \prec \{c\}$*
- *if $x \in \mathrm{state}(p)$ and $\hat{x} \prec C$ and $c < \hat{x}$ then $c \prec \{c\}$*
- *if $c_1 \prec C$ and $C_p \models c_1 = c_2$, then $c_2 \prec C$*
- *if $c_1 \prec C_1$, $c_2 \prec C_2$ and $C_p \models d = c_1 \vee c_2$ or $d = c_1 \wedge c_2$*

then $d \prec C_1 \cup C_2$

The clocks of p are computable iff for all bound signals $x \in X_p$ of p and all $c \leq x$ there exists C such that $c \prec C$.

Example 4 *Let us recall the graph of the switch in example 3. We observe that the clocks $\hat{s}$ and $\neg\delta_i$ cannot be computed.*

$$(x_1, x_2) = \mathsf{switch}(y_1, \delta_1, y_2, \delta_2) \stackrel{\text{def}}{=} \begin{pmatrix} \mathsf{toggle}(s, r) \,|\, x_1 = \mathsf{route}(s, \delta_1, y_1, y_2) \\ \big| \, x_2 = \mathsf{route}(s, \delta_2, y_2, y_1) \end{pmatrix} / s$$

We need to synchronize $\hat{s}$ to some input and export the signals δ_i. In doing so, we observe that r is redundant and that δ can be defined by the δ_i.

$$\mathsf{toggle}(s, \delta_1, \delta_2) \stackrel{\text{def}}{=} \begin{pmatrix} (s = (\,\mathsf{not}\, t \,\mathsf{when}\, \delta) \,\mathsf{default}\, (t \,\mathsf{when}\, \mathsf{not}\, \delta) \\ \big| \, (t = s \,\mathsf{pre}\, \mathsf{true}\,) \,|\, (s \,\mathsf{sync}\, \delta \,\mathsf{sync}\, (\delta_1 \,\mathsf{default}\, \delta_2)) \\ \big| \, ((\,\mathsf{when}\, \delta) \,\mathsf{sync}\, (\,\mathsf{when}\, \delta_1 \,\mathsf{default}\, \mathsf{when}\, \delta_2)) \end{pmatrix} / r\delta$$

Each δ_i can be seen as a flag coming along with y_i to decide whether to toggle the switch independently of the other (y_j, δ_j) pair.

$$x = \mathsf{route}(s, \delta, y, z) \stackrel{\text{def}}{=} ((x = (y \,\mathsf{when}\, s) \,\mathsf{default}\, (z \,\mathsf{when}\, \mathsf{not}\, s)) \,|\, (\delta \,\mathsf{sync}\, y))$$

Data-flow analysis relates the vertices of a graph G by a scheduling relation $g \gg h$. It is defined iff the name defined by g can eventually be used by h. Two non-causal vertices are said independent, noted $g \lesseqgtr h$, and then they are either exclusive, written $g\#h$, synchronous, written $g \sim h$, or concurrent, written $g \diamond h$.

Definition 7 *Two signal vertices $g, h \in S_p$ are* causal*, written $g \gg h$, iff* $\mathrm{def}(g) \in \mathrm{use}(h)$, $\mathrm{fun}(h) \neq$ pre *and $C_p \models \hat{g} \wedge \hat{h} \neq 0$. They are* independent*, written $g \lesseqgtr h$, iff $g \not\gg h$ and $h \not\gg g$.*

Two independent vertices g and h are exclusive*, written $g\#h$, iff $C_p \models \hat{g} \wedge \hat{h} = 0$. They are* synchronous*, written $g \sim h$, iff $C_p \models \hat{g} = \hat{h}$.*

Two independent vertices g and h are either connected*, written $g\tilde{\#}h$, iff $g\#h$ or $g \sim h$, or* concurrent*, written $g \diamond h$, iff $\neg g\tilde{\#}h$.*

Definition 7 provides a complete structure for the vertices of a data-flow graph: two vertices are either causal or independent. Two independent vertices are either concurrent, synchronous or exclusive.

Example 5 *In the case of the switch, we have the following signal and clock vertices:*

$$S = \begin{cases} g_1^s \stackrel{\mathrm{def}}{=} (\delta \Rightarrow s = \neg t) \\ g_2^s \stackrel{\mathrm{def}}{=} (\neg\delta \Rightarrow s = t) \\ g^t \stackrel{\mathrm{def}}{=} \big(\hat{t} \Rightarrow t = s \text{ pre true}\big) \\ g_1^{x_i} \stackrel{\mathrm{def}}{=} \big(e_i^1 \Rightarrow x_i = y_i\big) \\ g_2^{x_i} \stackrel{\mathrm{def}}{=} \big(e_i^2 \Rightarrow x_i = y_j\big) \end{cases} \quad \text{and } C = \begin{pmatrix} \hat{s}{=}\hat{t} = \hat{\delta} \\ \delta{=}\delta_1 \vee \delta_2 \\ \neg\delta{=}\neg\delta_1 \wedge \neg\delta_2 \\ \hat{y}_i{=}\hat{\delta}_i \\ \hat{x}_i{=}e_i^1 \vee e_j^2 \\ e_i^1{=}\hat{y}_i \wedge s \\ e_i^2{=}\hat{y}_i \wedge \neg s \end{pmatrix}$$

We have the following relations among the signal vertices:

$$\begin{array}{cccccc} g_1^s\#g_2^s & g_1^{x_2}\#g_2^{x_2} & g_2^{x_1}\#g_1^{x_2} & g_i^s{\gg}g_1^{x_1} & g_i^s{\gg}g_1^{x_2} & g_1^{x_1}{\diamond}g_1^{x_2} \\ g_1^{x_1}\#g_2^{x_1} & g_1^{x_1}\#g_2^{x_2} & g^t{\gg}g_i^s & g_i^s{\gg}g_2^{x_1} & g_i^s{\gg}g_2^{x_2} & g_2^{x_1}{\diamond}g_2^{x_2} \end{array}$$

and p is schedulable according to the following definition.

Event grammars A notion of grammar establishes the duality between the automaton and the graph of a process p by linking these representations. The grammar M_p of a process p consists of signal vertices related by the connectors $\gg$, $\#$ $\sim$ and $\diamond$. It can be viewed as a refinement of data-flow graph G_p which make the event structure implied by the clock relations C_p explicit.

$$M, N ::= g \mid M \gg N \mid M\#N \mid M \sim N \mid M \diamond N \quad \text{grammar}$$

To construct the event grammar of a process, we make use of a few functions to manipulate the graph structure implied by the relation of causality. The immediate successors and predecessors of a vertex g in a graph G are noted $\mathrm{succ}_g(G)$ and $\mathrm{pred}_g(G)$ and their transitive closures $\mathrm{succ}_g^*(G)$ and $\mathrm{pred}_g^*(G)$, respectively. The minimal and maximal vertices of a graph g are noted $\min(G)$ and $\max(G)$. The neighbors of g in G are $\mathrm{next}_g(G) = \cup_{h \in \mathrm{pred}_g(G)} \mathrm{succ}_h(G)$.

$$\begin{array}{ll} \mathrm{pred}_g(G){=}\{h{\in}G \mid h{\gg}g\} & \min(G){=}\{g{\in}G \mid \forall h{\in}G,\ h{\not\gg}g\} \\ \mathrm{succ}_g(G){=}\{h{\in}G \mid g{\gg}h\} & \max(G){=}\{g{\in}G \mid \forall h{\in}G,\ g{\not\gg}h\} \end{array}$$

Definition 8 *The* grammar M *is defined from the signal vertices of a graph* S *by the function* $\mathrm{fork}(S)$. *We write* $\mathrm{fork}_g(S)$ *for the prefix of* g *in the grammar* $\mathrm{fork}(S)$.

$$\begin{aligned}\mathrm{fork}(S) = \;& \textit{let } \diamond_{i=1}^{m}(\tilde{\#}_{j=1}^{n_i} g_{i,j}) = (\max(S))_{\tilde{\#}} \\ & \textit{in } \diamond_{i=1}^{m}\left(\tilde{\#}_{j=1}^{n_i}\left(\mathrm{fork}\left(\mathrm{pred}^*(g_{i,j})\right)\right) \gg g_{i,j}\right)\end{aligned}$$

The partition $S_{\tilde{\#}} = (S_i)_{i=1}^{n}$ *of a set of independent signal vertices* S *into exclusive vertices is defined by* $S = \uplus_{i=1}^{n} S_i$ *and, for all* $0 < i, j \leq n$, *for all* $(g, b) \in S_i \times S_j$, $(g \tilde{\#} h \Leftrightarrow i = j)$ *and* $(g \diamond h \Leftrightarrow i \neq j)$.

A sequential component of M is a sub-grammar that does not contain a concurrency or synchrony relation (only $\gg$ or $\#$).

Example 6 *As an example, the grammar of the switch is* M_{switch} *and the prefix of* $g_1^{x_1}$ *is* $g^t \gg (g_1^s \# g_2^s)$. *The grammar* $g^t \gg (g_1^s \# g_2^s) \gg (g_1^{x_1} \# g_2^{x_1})$ *is a sequential component of* M_{switch} *and* $g^t\, g_1^s\, g_1^{x_1}$ *one of its threads.*

$$M_{\mathsf{switch}} \stackrel{\mathrm{def}}{=} g^t \gg (g_1^s \# g_2^s) \gg ((g_1^{x_1} \# g_2^{x_1}) \diamond (g_1^{x_2} \# g_2^{x_2}))$$

A thread T is a sequence of signal vertices that represents the possible scheduling of a transition. We call $|T|$ its length and T_i its ist element. The threads $\mathrm{join}(M)$ of a grammar M are constructed by structural induction

$$\begin{aligned}\mathrm{join}(g) &= g \\ \mathrm{join}(M \# N) &= \mathrm{join}(M) \cup \mathrm{join}(N) \\ \mathrm{join}(M \gg N) &= \{TU \mid (T, U) \in \mathrm{join}(M) \times \mathrm{join}(N)\} \\ \mathrm{join}(M \sim N) &= \mathrm{join}((M \gg N) \# (N \gg M)) \\ \mathrm{join}(M \diamond N) &= \mathrm{join}((M \sim N) \# M \# N)\end{aligned}$$

7 Decision procedures

The control and data-flow analysis of a process define an event structure represented by a grammar M_p that allows us criteria upon which a process p can be guaranteed to be weakly-endochronous.

Determinacy and schedulability First, we show that checking a process p deterministic and step independent amounts to the satisfaction of clock relations in C_p, some of which being related to the causal structure implied by its graph G_p.

Let (g, h) two signal vertices of a graph G_p. If $g \gg h$ then $(\hat{g} \wedge \hat{h}) \Rightarrow \mathrm{def}(g) \gg^* \mathrm{def}(h)$. If $e_1 \Rightarrow x \gg^* y$ and $e_2 \Rightarrow x \gg^* y$ then $(e_1 \vee e_2) \Rightarrow x \gg^* y$. If $e_1 \Rightarrow x \gg^* y$ and $e_2 \Rightarrow y \gg^* z$ then $(e_1 \wedge e_2) \Rightarrow x \gg^* z$.

Definition 9 *A process* p *is* schedulable *iff* $e \Rightarrow x \gg^* x$ *implies* $C_p \models e = 0$ *for all* x. *A process* p *is* predictable *iff its clocks are computable and all distinct vertices* $g \neq h$ *of* S_p *such that* $\mathrm{def}(g) = \mathrm{def}(h)$ *are exclusive* $g \# h$.

Independence Second, grammars provide the necessary framework to finitely explore the state-space of a process and define decision procedures for the properties of clock and choice-independence. We write $S|_I$ for the restriction of S on vertices that have used or defined variables in I.

Definition 10 *Thread T is compatible with U on I, written $T \triangleright_I U$, iff for all $x \in I$ and $0 < i \leq |T|$, there exists $0 < j \leq |U|$ s.t. $\hat{T}_i \Rightarrow \hat{x}$, $\hat{U}_j \Rightarrow \hat{x}$ and $\hat{T}_i \wedge \hat{U}_j \neq 0$.*

Two threads T and U are compatible *on I, written $T \bowtie_I U$, iff $T \triangleright_I U$ and $U \triangleright_I T$. Two grammars are compatible, written $M \bowtie_I N$, iff $T \bowtie_I U$ for all $(T, U) \in \text{join}(M) \times \text{join}(N)$. Two processes p and q are compatible $I = \text{vars}(p) \cap \text{vars}(q)$ iff $M_p \bowtie_I M_q$.*

Definition 11 *A process p is* conflit-free *iff, for all $g, h \in S = (S_p)|_{\text{vars}(p)}$,*

- *if $g \diamond h$ then for all $(g', h') \in \text{pred}_g(S) \times \text{pred}_h(S)$, $g' \neq h'$, $\text{use}(g') \cap \text{use}(h') \cap \text{vars}(p) = \emptyset$ and $g' \lesseqgtr h'$.*
- *if $g \# h$ and $\text{fork}_g(S) \bowtie_{\text{vars}(p)} \text{fork}_h(S)$ then there exists $(g', h') \in \text{next}_g(S) \times \text{next}_h(S)$, $g \sim h'$ and $h \sim g'$.*

A conflict-free process p is clock and choice independent: for instance, the grammar of the switch restricted to its free variables is conflict-free: $(g_1^{x_1} \# g_2^{x_1}) \diamond (g_1^{x_2} \# g_2^{x_2})$. At last, Definition 12 summarizes the above and defines a notion of weak controllability that is sufficient for a process to be weakly endochronous and a pair of processes weakly isochronous.

Definition 12 *A process p is* weakly controllable *iff it is schedulable, predictible and conflict-free. Two processes p and q are* mutually controllable *iff p and q are compatible and $p|q$ is schedulable.*

Our main result is in accordance to this definition.

Theorem 2 *If p is weakly controllable then A_p is weakly endochronous. If p and q are weakly and mutually controllable then A_p and A_q are weakly isochronous.*

8 Synthesis

We define the procedure for synthesizing a weakly-endochronous automaton starting from a weakly controllable process p. The function $\text{Comp}(p)$ uses four basic code generation operators to construct an automaton meeting the property of weak endochrony. It is additionally associated with sequential C-like pseudo-code simulating the behavior of the generated automaton.

Synthesis operators We use four basic operations on nodes A and B to assemble the automaton of a process p. The depiction $\circ \text{—} A \longrightarrow \circ$ stands for a transition system A whose initial and final states are exposed left and right and can be unified to others as, e.g., in $\circ \text{—} A \longrightarrow \circ \text{—} B \longrightarrow \circ$ where the final state of A is the initial state of B.

A node A can be decomposed into its activation clock c_A and two transition systems R_A and W_A (each of them with one initial state and one final state): R_A is used to read the signal and define

the clock c_A and W_A to write the signal defined by A. Pseudo-code is equivalently decomposable into $\texttt{A}=^{\text{def}}\texttt{R}_\texttt{A}$; `if` $\texttt{c}_\texttt{A}$ `then` $\texttt{W}_\texttt{A}$.

$$A \stackrel{\text{def}}{=} \circ - R_A \rightarrow \circ \xrightarrow{?c_A=1} \circ - W_A \rightarrow \circ \quad (?c_A=0)$$

The function $\mathrm{And}(A, B)$ serializes the execution of B after A. Its pseudo-code is $(R, c, W) = (R_A, c_A, \texttt{W}_\texttt{A}; \texttt{B})$.

$$\mathrm{And}(A,B) = \circ - R_A \rightarrow \circ \xrightarrow{?c_A=1} \circ - W_A \rightarrow \circ - B \rightarrow \circ \quad (?c_A=0)$$

The function $\mathrm{Or}(A, B)$ either performs A or B upon the value of the exclusive clocks c_A and c_B. The sequential pseudo-code simulating its behavior consists of $R = \texttt{R}_\texttt{A}; \texttt{R}_\texttt{B}$, $c = \texttt{c}_\texttt{A} \mid\mid \texttt{c}_\texttt{B}$ and $W =$ `if` $\texttt{c}_\texttt{A}$ `then` $\texttt{W}_\texttt{A}$ `else if` $\texttt{c}_\texttt{B}$ `then` $\texttt{W}_\texttt{B}$.

$$\mathrm{Or}(A,B) = \circ - R_A R_B \rightarrow \circ \begin{cases} ?c_A=1 ?c_B=0 \rightarrow \circ - W_A \rightarrow \\ ?c_A=0 ?c_B=0 \rightarrow \\ ?c_A=0 ?c_B=1 \rightarrow \circ - W_B \rightarrow \end{cases} \circ$$

The function $\mathrm{Any}(A, B)$ executes none, either or all of two concurrent nodes A and B upon the availability of the clocks c_A and c_B: $\mathrm{Any}(A, B) = \mathrm{Or}(\mathrm{And}(A, B), \mathrm{And}(B, A))$.

Synthesis of a graph The synthesis of the automaton or pseudo-code for a graph consists of an inductive decomposition of its grammar and is defined by the function $\mathrm{Comp}(M)_X$.

$$\begin{array}{ll} \mathrm{Comp}(M \gg N) & = \mathrm{And}(\mathrm{Comp}(M), \mathrm{Comp}(N)) \\ \mid \mathrm{Comp}(M \# N) & = \mathrm{Or}(\mathrm{Comp}(M), \mathrm{Comp}(N)) \\ \mid \mathrm{Comp}(M \sim N) & = \mathrm{Any}(\mathrm{Comp}(M), \mathrm{Comp}(N)) \\ \mid \mathrm{Comp}(M \diamond N) & = \mathrm{Any}(\mathrm{Comp}(M), \mathrm{Comp}(N)) \\ \mid \mathrm{Comp}(g) & = (\mathrm{Rcv}(g), \mathrm{Clk}(g), \mathrm{Snd}(g)) \end{array}$$

The auxiliary functions Rcv and Snd handle the actions of reading input clocks and writing the output signal of a vertex g. The function $\mathrm{Rcv}(g)$ defines the clock of a signal vertex g by the set C of clocks satisfying $\hat{g} \prec C$ and by the Boolean function f satisfying $C_p \models \hat{g} = f(C)$. Its pseudo-code is $\texttt{All}(\texttt{Chk}(\texttt{C}))$; $\texttt{snd}(\texttt{c}_\texttt{x}, \texttt{f}(\texttt{v}_{\texttt{c}_1} .. \texttt{v}_{\texttt{c}_n})$ where the presence of an input signal x is defined by the boolean variable c_x.

$$\mathrm{Rcv}(g) = \circ - (\mathrm{All}(\mathrm{Chk}(c) \mid c \in C)) \rightarrow \circ \xrightarrow{!c_x = f(v_{c_1} .. v_{c_n})} \circ \quad \begin{array}{l} \text{where } x = \mathrm{def}(g) \\ \text{and } \hat{x} \prec \{c_{1..n}\} \\ \text{and } C_P \models \hat{x} = f(c_{1..n}) \end{array}$$

The auxiliary function $\mathrm{Chk}(c)$ defines the label l that checks a the presence of a clock c and defines the variable v_c. Pseudo-code, depicted on the right, uses the function `chk(x)`, that returns the

presence/absence status of the signal x, and the function `rcv(x)`, that reads a value from the input buffer of x. To allow for reading signals multiple times and writing them only once, read signals are marked and marked signals are only flushed upon finalization of the reaction. By contrast, written signals are flushed immediately in the buffer to allow for other processes to read them.

$$\begin{array}{l|l} \mathrm{Chk}(\hat{x}) \;\; =?c_x = v_{c_{\hat{x}}} & \mathtt{c_{\hat{x}}} = \mathtt{chk(x)} \\ \mid \mathrm{Chk}(x) \;\; =?x = v_x & \mathtt{v_x} = \mathtt{chk(x)\,?\,rcv(x)\,,\,0} \\ \mid \mathrm{Chk}(\neg x) =?x = \neg v_{\neg x} & \mathtt{v_{\neg x}} = \mathtt{chk(x)\,?\,!rcv(x)\,,\,0} \end{array}$$

The auxiliary function $\mathrm{All}(A, B)$ combinatorially schedules the read labels l_1 and l_2 and can be simulated by $\mathtt{l_1;\ l_2}$.

$$\mathrm{All}(l_1, l_2) = \quad \circ \xrightarrow{\begin{array}{c} l_1 \nearrow \quad \searrow l_2 \\ l_1 l_2 \\ l_2 \searrow \quad \nearrow l_1 \end{array}} \circ$$

The function $\mathrm{Snd}(g)$ defines the action of a signal vertex g. It first reads its used signals $y_{1..n}$ and then writes its defined signal x. The auxiliary function $\mathrm{Get}(x)$ is defined by $(?x = v_x)$ and has pseudo-code $\mathtt{v_x} = \mathtt{rcv(x)}$.

$$\mathrm{Snd}(c \Rightarrow x = f(y_{1..n})) = \; \circ - (\mathrm{All}\,(\mathrm{Get}(y_i)_{i=1}^{n})) \rightarrow \circ \xrightarrow{!x = f(v_{y_1}..v_{y_n})} \circ$$

Notes The synthesis of a delay statement $\mathrm{fun}(g) =$ pre is best depicted by pseudo-code. It defines a local variable `zx` initialized to v that stores the v_y upon finalization of a reaction to then load it as the previous value of y at the next reaction. The corresponding automaton is built by duplicating the transitions from and to the states corresponding to the possible values of x.

$$\begin{array}{l|l} \mathtt{bool\ zx = v;\ //\ initialize} & \mathtt{v_x = zy;} \\ \vdots & \mathtt{snd(x, v_x);} \\ \mathtt{v_y = rcv(y);} & \vdots \\ \vdots & \mathtt{zx = v_y;} \quad \mathtt{//\ finalize} \end{array}$$

Example 7 *The execution of the switch consist of statically scheduling the pseudo-code of its grammar:*

$$A_{\mathsf{switch}} \stackrel{\mathrm{def}}{=} \mathrm{And}\left(A^t, \mathrm{And}\left(\mathrm{Or}\left(A_1^s, A_2^s\right), \mathrm{Any}\left(\mathrm{Or}\left(A_1^{x_1}, A_2^{x_1}\right), \mathrm{Or}\left(A_1^{x_2}, A_2^{x_2}\right)\right)\right)\right)$$

The code of vertices consists of the following read, clock and write actions:

$$\begin{array}{llll|llll} R^t \stackrel{\mathrm{def}}{=} \mathtt{All}(\mathtt{R}^{\delta_1}, \mathtt{R}^{\delta_2}) & R_1^{x_i} \stackrel{\mathrm{def}}{=} \mathtt{All}(\mathtt{R}^{\hat{\mathtt{y}}_i}, \mathtt{R}^{\mathtt{s}}) & & & R^{\hat{y}_i} \stackrel{\mathrm{def}}{=} (\mathtt{c_{y_i}{=}chk(y_i)}) & & R^{\neg s} \stackrel{\mathrm{def}}{=} (\mathtt{v_{\neg s}{=}chk(s)?!rcv(s),0}) \\ R_1^s \stackrel{\mathrm{def}}{=} \mathtt{All}(\mathtt{R}^{\delta_1}, \mathtt{R}^{\delta_2}) & R_2^{x_1} \stackrel{\mathrm{def}}{=} \mathtt{All}(\mathtt{R}^{\hat{\mathtt{y}}_2}, \mathtt{R}^{\neg \mathtt{s}}) & & & R^{\hat{\delta}_i} \stackrel{\mathrm{def}}{=} (\mathtt{c_{\delta_i}{=}chk(\delta_i)}) & & R^{\delta_i} \stackrel{\mathrm{def}}{=} (\mathtt{v_{\delta_i}{=}chk(\delta_i)?rcv(\delta_i),0}) \\ R_2^s \stackrel{\mathrm{def}}{=} \mathtt{All}(\mathtt{R}^{\neg\delta_1}, \mathtt{R}^{\neg\delta_2}) & R_2^{x_2} \stackrel{\mathrm{def}}{=} \mathtt{All}(\mathtt{R}^{\hat{\mathtt{y}}_1}, \mathtt{R}^{\neg \mathtt{s}}) & & & R^s \stackrel{\mathrm{def}}{=} (\mathtt{v_s{=}chk(s)?rcv(s),0}) & & R^{\neg\delta_i} \stackrel{\mathrm{def}}{=} (\mathtt{v_{\neg\delta_i}{=}chk(\delta_i)?!rcv(\delta_i),0}) \end{array}$$

And last, the clock and write actions in the code of vertices is as follows:

$$\begin{array}{l|l|l|l} c_1^s \stackrel{\mathrm{def}}{=} \mathtt{v_{\delta_1}} || \mathtt{v_{\delta_2}} & c_1^{x_i} \stackrel{\mathrm{def}}{=} \mathtt{c_{y_i}} \&\& \mathtt{v_s} & W_1^s \stackrel{\mathrm{def}}{=} \mathtt{snd(s, !v_t)} & W_1^{x_i} \stackrel{\mathrm{def}}{=} \mathtt{snd(x_i, v_{y_i})} \\ c_2^s \stackrel{\mathrm{def}}{=} (\mathtt{v_{\neg\delta_1}} \&\& \mathtt{v_{\neg\delta_2}}) & c_2^{x_1} \stackrel{\mathrm{def}}{=} \mathtt{c_{y_2}} \&\& \mathtt{!v_s} & W_2^s \stackrel{\mathrm{def}}{=} \mathtt{snd(s, v_t)} & W_2^{x_1} \stackrel{\mathrm{def}}{=} \mathtt{snd(x_i, v_{y_2})} \\ c^t \stackrel{\mathrm{def}}{=} \mathtt{c_{\delta_1}} || \mathtt{c_{\delta_2}} & c_2^{x_2} \stackrel{\mathrm{def}}{=} \mathtt{c_{y_1}} \&\& \mathtt{!v_s} & W^t \stackrel{\mathrm{def}}{=} \mathtt{v_t = zs; snd(t, v_t)} & W_2^{x_2} \stackrel{\mathrm{def}}{=} \mathtt{snd(x_i, v_{y_1})} \end{array}$$

One may consider an implementation of the above using a generic scheduler, in a way related yet more general than that proposed in [11]. Our method allows execution of n separately compiled sequential components defined by arrays of guards R, clocks c and actions W by implementing the function $\mathrm{Any}(n, R, c, W)$ *as follows:*

```
Any(n, R, c, W) {
  i = 0;
  for(i = 0; i < n; i++)
    d[i] = 0;
      do {
      ⋮
⋮
a = 0;
for(i = 0; i < n; i++) {
  if !d[i] {
    (R[i])();
    if c[i] {
    ⋮
}
                ⋮
                (W[i])();
                d[i] = 1;
                a = 1;
        }}}} while a
```

Weak isochrony for free In the proposed code generation scheme, we observed that separately compiled vertices and sequential components could be executed in sequence, statically scheduled, dynamically scheduled or distributed without affecting the meaning of the process provided the static properties of weak-controllability to be satisfied and hence the control-structure of the process not broken.

In particular, distributed code generation schemes previously proposed seamlessly apply to the present scheme. These technique mainly consist of optimizing clock exchanges between communicating distributed processes. We observed that the property of weak-controllability requires few synchronizations for a process to form a compilation or distribution unit, much fewer than endochrony (Section 2).

Nonetheless, our simple and modular code generation scheme could of course be greatly optimized, for instance by keeping track of the clock defined and signal read along a given thread. However, and to that respect, the aim of the present article was primarily to demonstrate feasibility.

9 Conclusions

The synchronous paradigm was originally proposed as a conceptual framework to ease the computer-assisted design of embedded systems by abstracting timing issues with an idealized mathematical models in which communication and computation take no time. In this framework, timing properties of causality and synchrony can be represented in relational algebra to ease compilation and verification. While specification and symbolic verification are thereby greatly simplified, implementation issues and physical mapping remain equally complex tasks and a conceptual gap apparent.

To bridge this gap, the topics of desynchronization proposes models and algorithms to map the synchronous and functional model of a system on its virtual execution architecture by imposing strong synchronization constraints to local and synchronous modules and incurs an equally strong synchronization scheme to global communications that, in the end, maximizes both bandwidth and latency.

The micro-step automata theory of Potop et al. and the data-structures and algorithms presented in this article allow us to attack the central issue of locally minimizing synchronization constraint and preclude optimized global communication synthesis schemes to be accordingly defined.

References

[1] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, v. 163. Academic Press 2000.

[2] 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*. IEEE Press, 2003.

[3] A. Benveniste, P. Caspi, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Embedded Software Conference*. Springer, 2003.

[4] C. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. The theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 20(9). IEEE Press, 2001.

[5] P. Caspi, A. Girault, D. Pilaud. Distributing Reactive Systems. International Conference on Parallel and Distributed Computing Systems. ISCA, 1994.

[6] P. Caspi, C. Mazuet, and N. Reynaud. About the design of distributed control systems: the quasi synchronous approach. In *International Conference on Computer Safety, Reliability and Security*. Springer, 2003.

[7] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*. World Scientific, 2003.

[8] N. Lynch and E. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, v. 82(1). Academic Press, 1989.

[9] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specification. In *Application of Concurrency to System Design*. IEEE Press, 2005.

[10] Talpin, J.-P, Le Guernic, P. An algebraic theory for behavioral modeling and protocol synthesis in system design In *Formal Methods in System Design*. Springer, 2005.

[11] Weil, D., et al. Efficient compilation of Esterel for real-time embedded systems. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, 2000.