Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 146 (2006) 151–167
www.elsevier.com/locate/entcs

A Survey of Desynchronization in a Polychronous Model of Computation

Julien Ouy[1]

IRISA, Université de Rennes I, France

Abstract

The *synchronous hypothesis* arose in the late Eighties as a conceptual framework for the computer-aided design of embedded systems. Along with this framework, the issue of desynchronization was simultaneously raised as the major topic of mapping the ideal communication and computation model of synchrony on realistic and distributed computer architectures.
The aim of the present article is to survey the development of this topics in the particular yet promising model of one of the prominent environments that were build along these principles: Signal and its polychronous (synchronous multi-clocked) model of computation, before to give some hints and ideas about ongoing research addressing this issue.

Keywords: Synchronous programming, desynchronization, GALS design, endochrony and isochrony

1 Introduction

The synchronous paradigm[3] has been 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, the timing properties of causality and synchronization are represented in relational algebra and to ease compilation and verification purposes. While the specification and verification of a system are greatly simplified, the actual code generation and physical mapping remain equally complex tasks [2].

[1] Email: Julien.Ouy@irisa.fr

doi:10.1016/j.entcs.2005.05.040

In particular, the synchronous hypothesis requires all components of a system to share a common notion of time. However, getting all components of a distributed system to share is most of the time impossible and, in the case of hardware implementations, an additional issue of wires length incurs additional constraints to already hard to manage timing issues, thereby compromising the ideal assumption of timeless computations and communications made during the functional specification of the system.

One conservative solution to this issue is to enforce resynchronization of distributed components all the way by making extensive use of synchronization protocols that clock the system at the pace of its slowest component and at the cost of efficiency. More elaborated methods are developed, especially in circuit design, that consist of replacing the clock distribution by a handshaking network[11] or by generic wrappers ensuring latency insensitivity to the designed system[8].

In embedded software design, however, the only way to go is with desynchronization: to map the synchronous model of a system on a virtual architecture consisting of several synchronous components communicating with asynchronous channels: a GALS architecture. In this survey, we study the issue of desynchronization in the particular context of a multi-clocked synchronous model of computation by considering the simple yet challenging example of the crossbar switch and study different method to implement it on a globally asynchronous and locally synchronous architecture.

2 Position of the problem

We consider the synchronous modeling and implementation of a simple crossbar switch depicted in figure 1. The switch consists of three input signals: two data signals x and y and one control signal r (the reset). It has two output signals a and b.

As in [14], the switch can be specified using guarded commands in the formalism of synchronous transition systems of Pnueli. The system has an internal state e that is initially set to 1 and toggled upon the presence of the reset signal r: if r is present then e equals the negation of its previous value, noted $e\$$

$$\mathsf{e}^0 = 1 \mid \mathsf{r} \Rightarrow \mathsf{e} = \neg \mathsf{e}\$$$

If $\mathsf{e}{=}0$, the output a is connected to the input y and b to x, and when $\mathsf{e}{=}1$ a

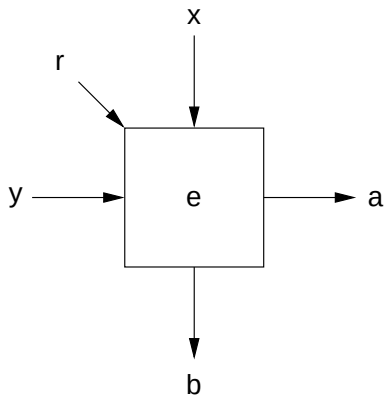

Fig. 1. The switch

is connected to x and b to y.

$$| \hat{x} \wedge e \Rightarrow a = x \qquad | \hat{y} \wedge \neg e \Rightarrow a = y$$
$$| \hat{y} \wedge e \Rightarrow b = y \qquad | \hat{x} \wedge \neg e \Rightarrow b = x$$

Asynchrony

In an asynchronous environment no clock can be defined and no absence sensed. Signals are ordered sequence of values, behaviours are tuples of signals and processes are sets of behaviours. The composition of two processes consists of the unification of all signal shared by the processes.

This paradigm is much closer to real architectures than synchrony. It only requires communication channels to respect the condition that an ordered stream of data sent reaches its destination in the same order.

We cannot make a direct implementation of the switch in an asynchronous environment as the interleaving of r and x or y may produce unpredictable results like in figure 2. The trace in the upper-middle of the figure is obtained if r is read before x and y. The trace in the lower-middle is possible if r is read after one value of x and y. In the trace on the right, r is read after only one value of x

Strict synchrony

The model of strict synchrony is the simplest and the most general. It is used in system on chips design and in scientific engineering (for a discrete model of time) and in automata theory. In this model, a state x assigns a unique value to each variable of the system $v \in V$. A behaviour is a sequence of states $\sigma = x_1, x_2, \ldots$ and a process a set of behaviour. A signal is a sequence

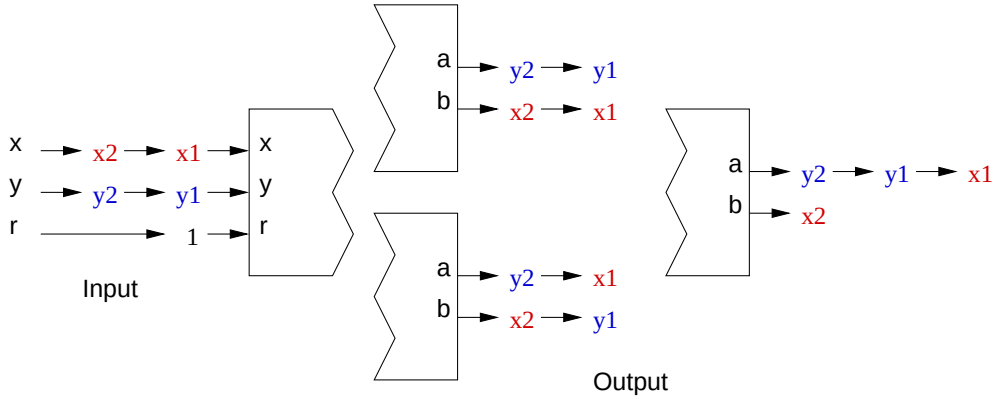

Fig. 2. Unpredictability by brute-force desynchronization

of values $\sigma_v = v(x_1), v(x_2), \ldots$ for $v \in V$ given. All signals are indexed by the same set of integer, all behaviour are synchronous and ruled by the same clock. Composition of processes is then defined as the intersection of the sets of behaviours.

$$P||P' = P \cap P'$$

This paradigm is followed by programming languages such as Lustre. In this model, our switch has two different states : $e = 0$ and $e = 1$, each relative to a basic transition relation.

$$\begin{array}{ll} e = 1 \Rightarrow \neg r \Rightarrow a := x & e = 0 \Rightarrow \neg r \Rightarrow b := x \\ e = 1 \Rightarrow \neg r \Rightarrow b := y & e = 0 \Rightarrow \neg r \Rightarrow a := y \\ e = 1 \Rightarrow r \Rightarrow b := x & e = 0 \Rightarrow r \Rightarrow a := x \\ e = 1 \Rightarrow r \Rightarrow a := y & e = 0 \Rightarrow r \Rightarrow b := y \\ e = 1 \Rightarrow e' := \neg r & e = 0 \Rightarrow e' := r \end{array}$$

If we feed the system with the the signals $x = x_1, x_2, x_3, x_4$, $y = y_1, y_2, y_3, y_4$, and $r = 0, 1, 0, 0$ it deterministically reacts as in figure 3, producing $a = x_1, y_2, y_3, y_4$ and $b = y_1, x_2, x_3, x_4$.

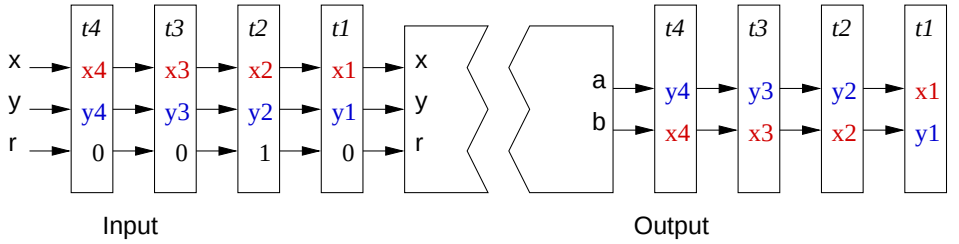

Fig. 3. Input and output of the switch with strictly synchronous signals

Synchrony

The strict synchronous paradigm can be extended by augmenting every domain of data with the special value $\perp$ to mean absence. The non-informative value $\perp$ is equivalent to the absence of value of a signal at a given time. Synchronous programs are able to interpret absence as control information and the Signal language implements this paradigm[13]: multiclocked synchrony.

In Polychrony, a state can assign a non-informative value to any variable, a synchronous behaviour is a sequence of synchronous states, a synchronous process is a set of synchronous behaviours. A synchronous signal is a sequence of informative and non-informative values. Composition of process is still defined by the intersection of sets of behaviours. Strictly synchronous processes can be considered as synchronous one in which every value are always "present".

A reaction is called silent if all variables are absent at the time of a reaction and if every component of a system has its own local clock consisting of the sequence of its non-silent reactions.

In the example of the figure 4, the left component (P) and the upper right component (P1) of the system share the same local clock, they have height reactions during the execution of the example. Components P2, P4 and P8 have respectively four, two and one time clock during the same time. Their local clocks are slower.

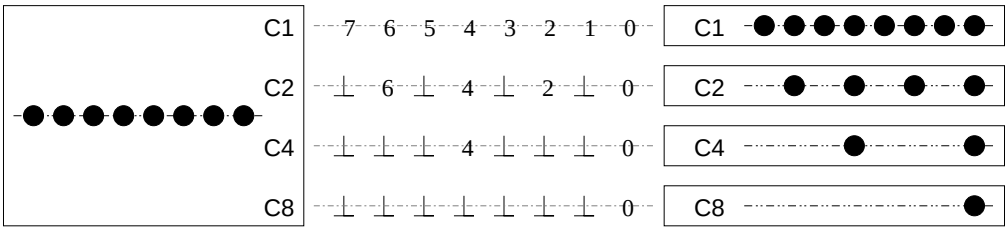

Fig. 4. Local clocks of different components of a system

In synchronous mode and at each instant, the switch produces a value only if there is an available incoming value. The r signal can now be considered as an alarm event, running only when needed and no longer emitting useless null vale. The figure 5 shows an example with synchronous signals.

Global asynchrony and local synchrony

The Globally Asynchronous Locally Synchronous (GALS) [9] paradigm is a combination of the asynchronous and the synchronous paradigm. It consists of several synchronous modules that communicate asynchronously. In circuit design, it relates to the modeling of small synchronous blocks communicating asynchronously. In software design, it relates to finite automatons

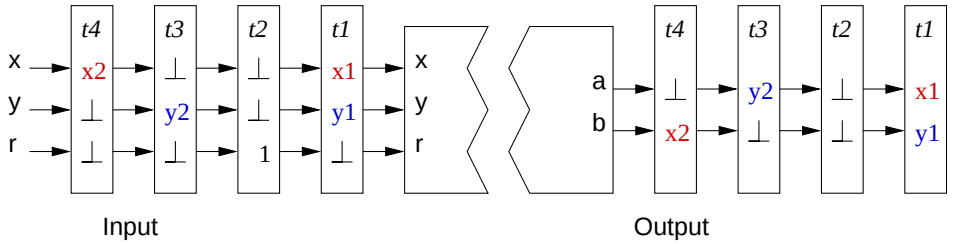

Fig. 5. Input and output with synchronous signals

that communicate with registers. There are many different paradigms for the implementation of GALS systems and there are many ways to obtain them. Some of them will be presented in this survey.

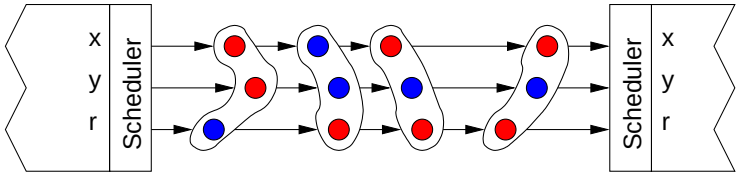

Fig. 6. Synchronous signals in GALS

3 Desynchronization

3.1 Latency Insensitive Systems

In [6], a general method to implement strictly synchronous distributed systems over non synchronous channels is presented. With an appropriate protocols and minimal assumptions on the modules behaviours, the theory of latency insensitive protocols guarantees the correct behaviour of the system independently of the channels latency.

Those latency-insensitive systems are composable using the synchronous assumption (they must share the same clock) but if correctness is not affected by latency, performances are: different modules cannot have different execution rate and delay between successive modules affects global computation time.

Latency insensitive protocols use a system of buffers at input and output of the synchronous systems: each signal has an associated FIFO buffer that is filled until every buffer has a value. When this happens, a reaction is fired and completed and the output buffers can deliver one value for each processed signal.

In this model, processes need only be stallable, meaning that time between two reactions is not fixed and that a reaction may occur at any time. This

recreates strict synchrony.

3.2 Automatic distribution in Post-compilation

The Ocrep tool [7] implements a post-compiler for the synchronous languages Lustre and Esterel. This post-compiler automates the distribution of an OC program (a finite automaton compiled from Lustre or Esterel) on a specific distributed architecture. The source program is compiled once and distributed code is generated for every component of the system. Each distributed component runs its own share of the code and communications are implemented by FIFOs to respect strict synchrony. Like with latency insensitive protocols, every component in Ocrep retrieves available inputs, performs its share of processing, signals the end of its reaction and waits upon a global synchronization signal before to perform the next reaction.

The Ocrep distribution has been proved correct in [5]: distributed and centralized program are equivalent.

The same approach is also supported for SC programs [12] (dual form of a sequential control code and a table of external actions) named Screp. SC compilation is for the moment only available for Esterel and permits to avoid the combinatorial explosion due to the automaton nature of the OC support.

3.3 Isochrony and Endochrony

A solution to desynchronization consists of recreating non-strict synchrony by adding signals that clock desynchronized signals. Each signal receives an associated boolean signal telling whether it is present or absent at a given instant: its models its clock. This technique permits to implement the criterion of *Isochrony*[1].

Definition 3.1 Two processes are isochronous if there exists no pair of reactions that are not synchronisable.

In Signal, the technique of associating a Boolean clock with a signal is called *booleanization*. At each tick of a given clock C, we check if a signal s defined in that clock domain is present or not. If it is, then the clock signal of s is written $C_s = 1$ and, if not, $C_s = 0$.

The main issue with this approach lies in the choice of the reference clock C. First we consider the clock of the emitting system.

In Figure 7, left, we have three synchronous signals, x,y and r. x and y carry data and the signal r is an event, i.e. a signal that is always true when present. In the middle, the same data are displayed without presence/absence values. The clock signals r carry sufficient information to reconstruct the

synchronization relation between x and y. On the right, the data are put in asynchronous channel.

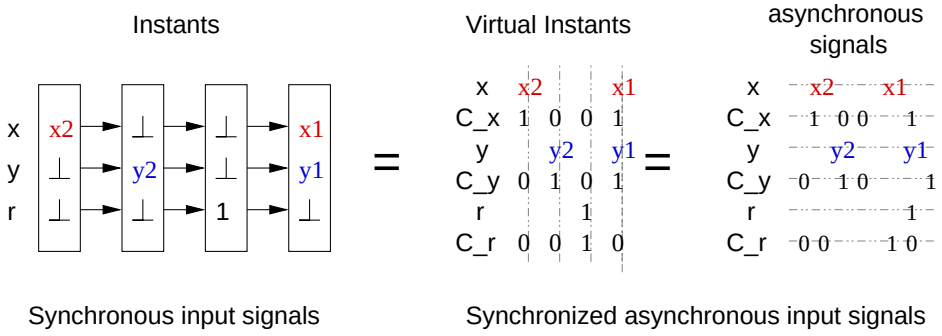

Fig. 7. Equivalent signals in synchronous and asynchronous form

We observe that the signal of r is not useful anymore. The information it carries is also carried by its clock. C_r replaces it.

With asynchronous signal carrying data and clocks, a unique synchronous sequence of data is reconstructed. The system is endochronous This property of *endochrony* by Benveniste [1] is defined by:

Definition 3.2 A process is *Endochronous* if presence/absence of all variables can be inferred incrementally from already known values and variables present in the system.

Input clocks in an endochronous system have sufficient relations to infer the presence/absence status of all signals of the system at all time. The property *Endochrony* actually authorizes a more efficient solution in that the use of a master clock (one of the inputs) is just a choice and is not actually required formally.

Endochronous systems are rather close from Carloni's latency-insensitive systems: schedulers that process inputs and reconstruct synchronous flows while allowing the use of absence value, yielding some concurrency not available in strictly synchronous systems. Here, instants can be reconstruct as soon as every clock signal and *present* data signals have reach destination.

The main problem is that this solution does not carry over to composition (of endochronous processes). If two modules are ruled by different clocks, it is impossible to build a global process, reading their respective asynchronous signals and clocks. In fact, the problem comes from the impossibility to merge two different relative instants of two endochronous processes. Information is lacking to resynchronize the flow of both signals.

Composition is only possible if the two processes have the same clock or if the clock of one is a fraction of that of the other, and then the processes may

need a protocol to initially synchronize their clocks and keep them synchronized.

In figure 8, we notice that synchronous or asynchronous signals are not composable as a common process. The problem is due to the lack of synchronization information between signals of the two process since their clocks are not related.

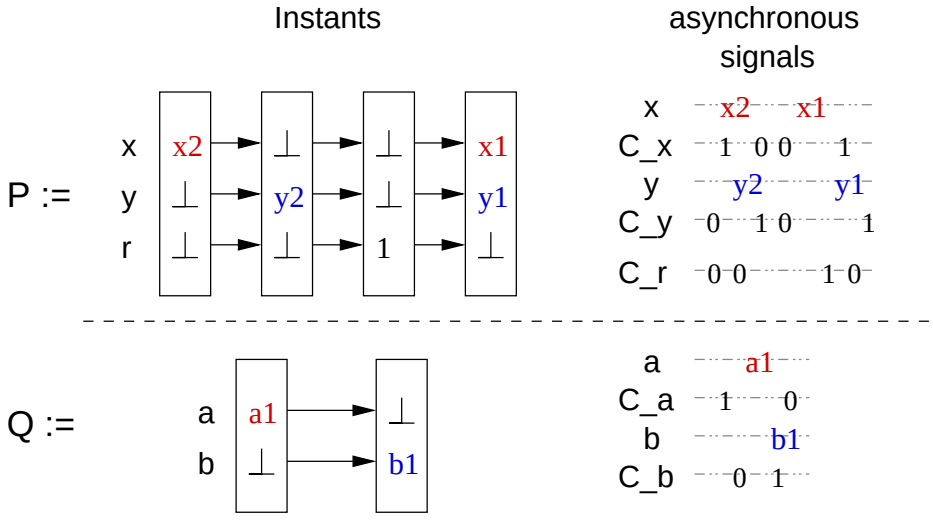

Fig. 8. Two signals in their both form

The signals from different processes may have many kind of synchrony relations:

- The processes communicate together and share a clock.
- Processes do not communicate but share the same clock from a third process.
- Processes do not communicate and signals are not related.

For the two first cases, an external scheduler can reconstruct the instants. For the last one, we must permit our processes to be polychronous: two signals with different clocks / instants must be able to coexist in the same process.

3.4 Differential clocks and polychrony

In a polychronous process (or in an asynchronous one), the flow of unbounded incoming data can not be known. Figure 9 displays two possible clock references in a system, data can come, and be read, in different and unrelated orders. However, the process must be able to reconstruct the same behaviour whatever this order is.

With multi-clocked synchronous communication capabilities, two process

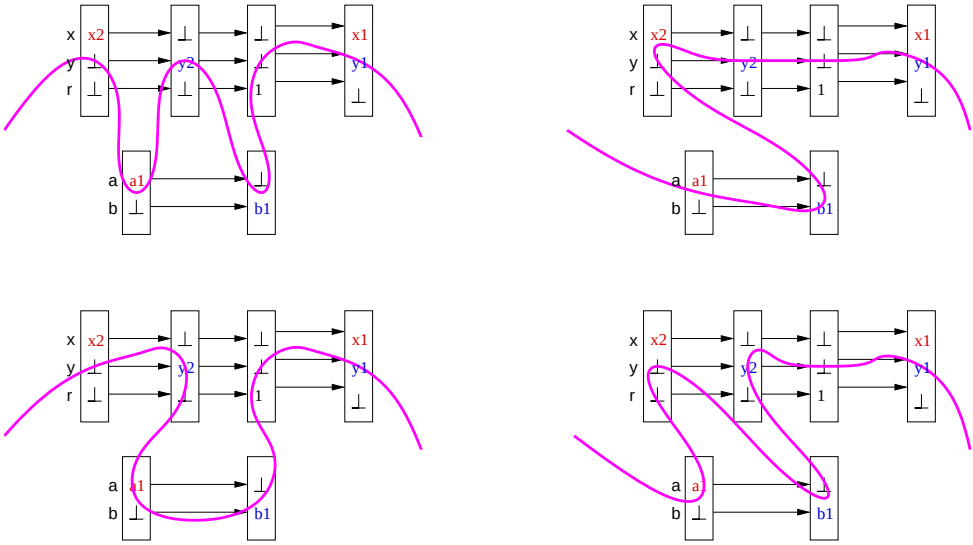

Fig. 9. Different execution flows for the same data

can interact at the pace of another clock, a differential clock, without having to agree upon a common or shared clock. The synchronous language Signal supports the specification of such polychronous processes. In the Signal transformation-based compilation process, clocks will remain unrelated as long as the user specifies some synchronization relation as needed. For instance, Signal supports the following specification of the crossbar switch:

```
process switch = (? event r; boolean x, y ! boolean a, b)
    (| e := not e$ init true when r default e$ init true
     | a := (x when e) default (y when not e)
     | b := (y when e) default (x when not e)
     |) where boolean e;
end;
```

The signals r, x and y are the declared inputs of the switch and the signals a and b its outputs. Signal e is a local variable that remembers the state of the switch. It is defined with its former value and the actual value of r. a and b are defined with x or y depending of e.

The Signal compiler decomposes specifications into a format called Dc+ that defines a directed graph obtained by causality analysis and a clock hierarchy obtained by analysing synchronization relations.

From the Dc+ format, a system of equation on clocks is constructed. The solution of this system defines inclusion relations between clocks. The clocks are organized in a hierarchy according to this inclusion relation.

The main phase of the compilation of a Signal program is the clock calculus. First, every clock involved in the program are collected: they can be

- the clock of an input signal of the program, present when the signal is

available

- the positive and negative sampling of a boolean condition, they are present when the boolean is present and respectively true or false
- the sum of two clocks, present when at least one of the clocks is present
- the product of two clocks, present when both clocks are present
- the complementary of a clock with reference to an other clock, present when the second is true and the first is false.

Clock trees are build by placing every clock of the program under their predecessor.

This results a forest of trees of which every root is a free clock. If only one root is present (one tree), this root is the main clock of the program, the program is endochronous. If not it is exochronous, however it is still possible to construct either a multi-clocked system in which each root clock yields a process for parallel execution or a boolean single-clocked program for sequential execution.

In our example standard dC+ compilation produces a Signal program with the same interface

```
( ? event r; boolean y1, y2;
  ! boolean x1, x2; )
```

and bdC+ or C compilation produces an endochronous program that requires a main clock in input:

```
( ? boolean y1, y2, C_r, C_y1, C_y2, C_;
  ! boolean x1, x2; )
```

This transformation can be decompiled in Signal code as in the specification `switch_ABSTRACT` listed next. The clocks of the switch are computed and causality relations specified but clocks are not hierarchized one to the other: the clock hierarchy has no common root. This means that the switch specification is not endochronous but that the compiler can refine it into a C program that is indeed endochronous by synthesizing the missing master clock.

Here is an abstract of the Signal program in which we can read computations over clocks. In the first three lines is the signature of the program, in the following block of four lines containing “when” are the actions of the program (not detailed). The last block of twelve lines is the calculus of the clocks when actions occur.

```
process switch_ABSTRACT =
   ( ? event r;
       boolean x, y;
     ! boolean a, b;
   )
   spec
      (| (| {x --> a} when CLK_16
          | {y --> a} when CLK_59
          | {x --> b} when CLK_54
          | {y --> b} when CLK_31
          |)
       | (| (| CLK_x ^= x
             | CLK_x := ^CLK_x
             |)
          | (| CLK_16 := CLK_x ^* CLK_13 |)
          | (| CLK_y ^= y
             | CLK_y := ^CLK_y
             |)
          | (| CLK_31 := CLK_13 ^* CLK_y |)
          | (| CLK_b ^= b
             | CLK_b := CLK_31 ^+ CLK_38
             |)
          | (| CLK_a ^= a
             | CLK_a := CLK_16 ^+ CLK_24
             |)
          | (| CLK_54 := CLK_40 ^+ CLK_b |)
          | (| CLK_59 := CLK_26 ^+ CLK_a |)
          |) where event CLK_59, CLK_54, CLK_a,
             CLK_b, CLK_31, CLK_y, CLK_16, CLK_x;
             end
   |)
   pragmas Black_Box "switch"
   end pragmas
%switch_ABSTRACT%;
```

The generated C code displays this synthesized clock $C\_$ along which an arbitrary hierarchy is chosen. Apart from that clock, one notices that all subsequent computations are independent one from each other (a sequential scheduling order is chosen for the purpose of code generation).

Each computations start with a test of the clock: `if (C_)`, then the action of the specified clock is performed. Here only control code appends at clocks $C\_$, C_x and C_y but for the two last clocks C_b and C_a, actions are performed: `w_switch_b(b)` and `w_switch_a(a)`.

```
EXTERN logical switch_iterate()
{
    if (!r_switch_C_r(&C_r)) return FALSE;
    if (!r_switch_C_x(&C_x)) return FALSE;
    if (!r_switch_C_y(&C_y)) return FALSE;
    if (!r_switch_C_(&C_)) return FALSE;
    C_56 = C_r && C_;
    if (C_)
        {
        if (C_56) s = !XZX; else s = XZX_17;
        }
    s_63 = (C_ ? s : FALSE);
    C_68 = (C_ ? !s : FALSE);
    C_86 = C_x && s_63;
    C_98 = C_y && s_63;
    if (C_x)
        {
        if (!r_switch_x(&x)) return FALSE;
        }
    x_77 = (C_x ? x : FALSE);
    C_92 = C_68 && x_77;
    C_b = C_92 || C_98;
    if (C_y)
        {
        if (!r_switch_y(&y)) return FALSE;
        }
    y_71 = (C_y ? y : FALSE);
    C_89 = C_68 && y_71;
    C_a = C_86 || C_89;
    if (C_b)
        {
        if (C_98) b = y; else b = x;
        w_switch_b(b);
        }
    if (C_a)
        {
        if (C_86) a = x; else a = y;
        w_switch_a(a);
        }
    switch_STEP_finalize();
    return TRUE;
}
```

3.5 Weak Isochrony and Weak Endochrony

In our desynchronization model, we add the possibility to define a local synchronization relation between two signals without defining a synchronization scheme that is global to the system. This allows us to simply decompose a

polychronous system into smaller subsets of strictly synchronous computations.

Definition 3.3 Let x_1 and x_2 two signals of clocks C_{x_1} and C_{x_2} and assume a function $\sigma(x_1, x_2)$ that reconstruct the synchronization relation between x_1 and x_2. If x_1 and x_2 are not synchronisable then $\sigma(x_1, x_2) = \emptyset$.

Back to the example of the switch, we would like the signals x and y to be of different clocks but both synchronized to r. In our example, x and y are supposed to be frequent and to possibly come from independent sources (they are not synchronized) and r is supposed to be an interrupt, emitted by one of the two sources but interrupting both flows. In fact, a non-$\perp$ value of the signal r is always associated with a value on x and y and a non-$\perp$ value on x or y is always associated with a value on r like in the figure 10. Therefore, $\sigma(x, y) = \emptyset$, $\sigma(x, r) \neq \emptyset$ and $\sigma(y, r) \neq \emptyset$

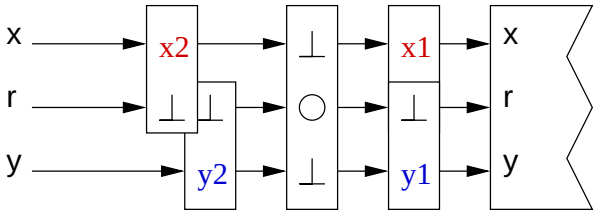

Fig. 10. Example of weakly-isochronous data flow

This rather weak constraint permits to lower the number of possible runs of the switch. Between two occurrences of a reset r, x and y can interleave without changing system's behaviour but each time a (non-$\perp$) value is emitted/received along r, the system synchronizes. We now want the system to be isochronous over the interfaces $I = \{\mathsf{x}, \mathsf{r}\}$ and $I' = \{\mathsf{y}, \mathsf{r}\}$

To obtain the corresponding desynchronization, we add clocks. Those clocks, called the differential clocks x/r and y/r of the system, are emitted as follows with the asynchronous channels x, y and r:

- when a value is emitted along the channel r only, an additional 0 is emitted along both x/r and y/r.
- when a value is emitted along x or y and nothing along r channel, an additional 2 is emitted alongx/r or y/r.
- when a value is emitted along r and either x or y, an additional 1 is emitted along either of the corresponding x/r or y/r.

The flow of the synchronous channels r, x and y may then be reconstructed the way the Petri net of figure 11 does. We have the expected guarantee that $\sigma(x, r) = \mathsf{x/r}$ and $\sigma(y, r) = \mathsf{y/r}$.

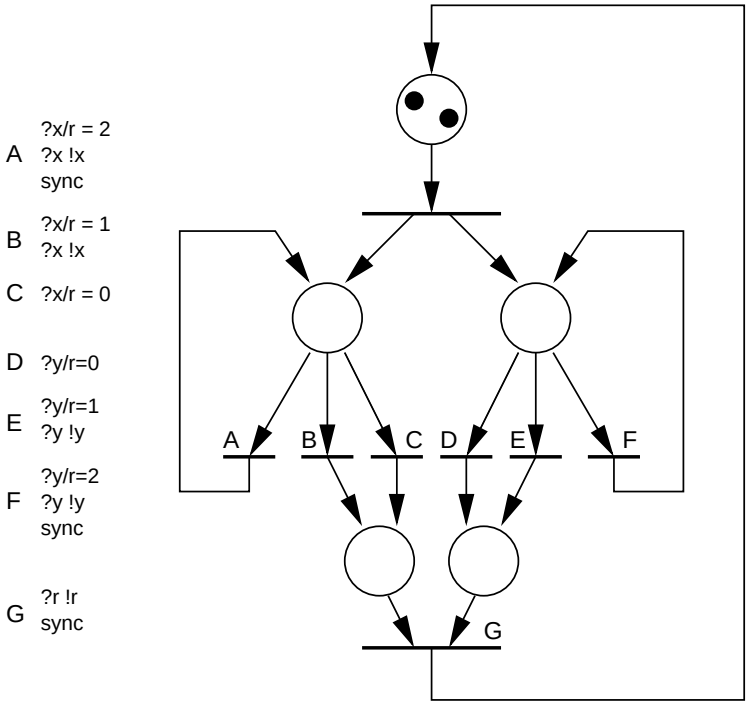

Fig. 11. Reconstruction of the desynchronized flows of r, x and y.

In this net, "?" means "read a value" and "!" means "write a value". Since the system is a synchronizer, every read is done on an asynchronous channel and every write on a synchronous one. The sync instruction means that an instant is finalized and a new one is starting.

The system reads the x and y streams as long as x/r and y/r gets the value 2. When one of them gets a 0 or a 1, it stops reading its associated stream until the other gets its own 0 or 1. Then, the r value is read, the intern state of the system (e) is modified and the system can go on. We notice that communicating the signal r is no longer useful: it can be reconstructed from x/r and y/r.

Weak Endochrony

In the aim of allowing for compositionally in the concurrent execution of synchronous modules, Potop et al. have defined a more liberal assumption than endochrony that allow for the asynchronous composition of the modules to meet determinism: weak-endochrony[4].

With *weak endochrony*, minimal reactions have to be constructed and synchronization clocks (σ) defined and associated to reactions. Once this is defined, one has to define a scheduling of the reactions that makes best usage

of these data. For maximized concurrency within a synchronous block, data must be used as soon as it is available and an order of execution not imposed unless explicitly specified. This amounts to decomposing a process into elementary blocks, each of them being activated upon availability of an input signal, each of them being inhibited by a conflicting signal (e.g. values 0 and 1 for a given differential clock). As shown in figure 12, communication between blocks is the same as communication between processes.

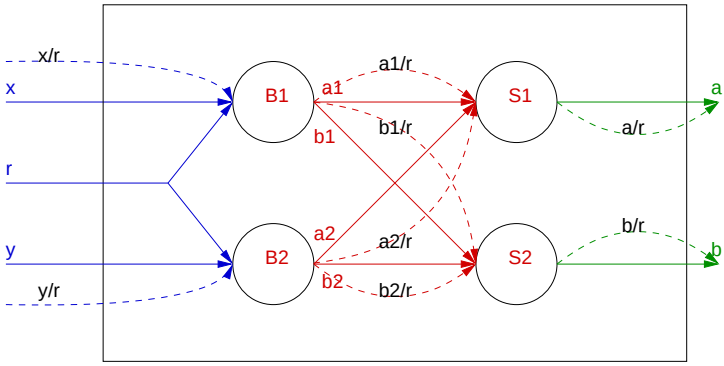

Fig. 12. decomposition of the switch process into blocks

Each block contains a single-clocked process and shares the inputs from the interface of the main process plus the outputs from other blocks. When more of one block contributes to define one signal, a synchronizer block is needed and, in this case, contributing signals have to be synchronizable ($\sigma(x_1, x_2) \neq \emptyset$).

Reaction Clock

The role of a reaction or activation clock is to define a set of instants. The clocks of signals involved in this reaction have to be sub-clocks of the reaction clock. With a synchronous language, like Signal, it is easy to merge two signals and hence build a reaction clock.

Since input signals are asynchronous, we need to specify each synchronization relation between them: if two input signals x_1 and x_2 need to be synchronized, their builder has to provide the synchronization function $\sigma(x_1, x_2)$. This can be a ternary signal, like in the example of the switch, this can also be a pair of binary signals clocked by the synchronous sum of the two signals and respectively true when one is present. Signals written by a block are always sons of its activation clock.

How to synchronize two signals produced by different blocks ?

If two output signals are build by different blocks then, most of the time, they are not synchronous. A solution to synchronize them is to activate both

blocks at the same clock, and the output signals then belong both to the set of instant of the common clock and can be synchronized to it.

We can also use a token to define writing turns. The blocks produce signals synchronized with the token signal. Every block can write when the token is present instant but each block can write between two tokens only every n tokens, where n is the number of blocks. A first turn is described on the figure 13. Synchronization only happen when token are present but writing turns permit to exclude overlapping conflicts.

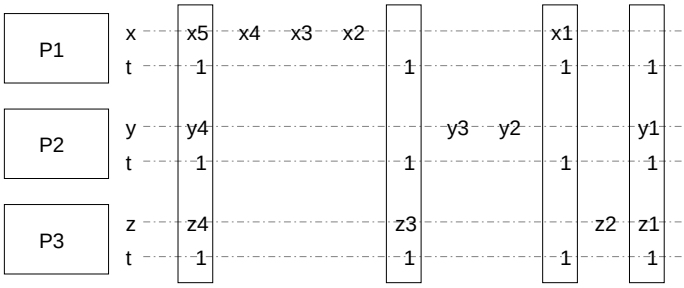

Fig. 13. Synchronization with a token

4 Conclusion

Solutions to the issue of desynchronized multi-clocked specifications are not unique. The generic and automated method of latency-insensitive design, section 3.1, is easy to develop since it does not require modifying the initial specification. This is at the cost of efficiency compared to other solutions such as endo-isochronous design, section 3.3, which achieves some better performances at the cost of compositionality. Weak endo-isochronous design, section 3.5, relaxes synchronization constraints enforced in the endo-isochronous approach and brings compositionality in a way that requires less resynchronization as in latency insensitive design.

References

[1] Benveniste, A., B. Caillaud and P. Le Guernic, *From synchrony to asynchrony*, Technical Report PI-1233, IRISA (1999).
URL ftp://ftp.irisa.fr/techreports/1999/PI-1233.ps.gz

[2] Benveniste, A., L. Carloni, P. Caspi and A. S. Vincentelli, *Heterogeneous reactive systems modeling and correct-by-construction deployment*, Technical Report PI-1549, IRISA (2003).
URL ftp://ftp.irisa.fr/techreports/2003/PI-1549.ps.gz

[3] Benveniste, A., P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic and S. Simone, *The synchronous languages twelve years later*, Proceedings of the IEEE, Special issue on Embedded Systems **91** (2003), pp. 64–83.

[4] Butucaru, D. P., B. Caillaud and A. Benveniste, *Concurrency in synchronous systems*, Formal Methods in System Design, Special Issue on formal methods for GALS design (2005).

[5] Caillaud, B., P. Caspi, A. Girault and C. Jard, *Distributing automata for asynchronous networks of processors*, JESA **31** (1997), pp. 503–524, research report Inria 2341.
URL http://www.inrialpes.fr/pop-art/people/girault/Publications/Jesa97

[6] Carloni, L. P., M. Mcmillan and A. L. Sangiovanni-Vincentelli, *The theory of latency insensitive design* (2001).

[7] Caspi, P., A. Girault and D. Pilaud, *Automatic distribution of reactive systems for asynchronous networks of processors*, IEEE Trans. on Software Engineering **25** (1999), pp. 416–427.
URL http://www.inrialpes.fr/pop-art/people/girault/Publications/Tse99

[8] Casu, M. R. and L. Macchiarulo, *A new approach to latency insensitive design*, in: *Proceedings of the 41st Annual conference on Design Automation (DAC-04)* (2004), pp. 576–581.

[9] Chapiro, D. M., "Globally-Asynchronous Locally-Synchronous Systems," Ph.D. thesis, Stanford University (1984).

[10] Colaço, J.-L., A. Girault, G. Hamon and M. Pouzet, *Towards a higher-order synchronous data-flow language*, in: G. Buttazzo, editor, *4th International Conference on Embedded Software, EMSOFT'04* (2004).

[11] Cortadella, J., A. Kondratyev, L. Lavagno, L. Lwin and C. Sotiriou, *From synchronous to asynchronous: an automatic approach*, in: *Proc. Design, Automation and Test in Europe (DATE)*, 2004, pp. 1368–1369.

[12] Girault, A. and C. Ménier, *Automatic production of globally asynchronous locally synchronous systems*, in: A. Sangiovanni-Vincentelli and J. Sifakis, editors, *International Workshop on Embedded Software, EMSOFT'02*, LNCS **2491** (2002), pp. 266–281.
URL http://www.inrialpes.fr/pop-art/people/girault/Publications/Emsoft02

[13] Maffeis, O. and P. Le Guernic, *Distributed implementation of SIGNAL: Scheduling and graph clustering*, Lecture Notes in Computer Science **863** (1994).

[14] Talpin, J.-P. and P. L. Guernic, *An algebraic theory for behavioral modeling and protocol synthesis in system design*, Formal Methods in System Design, Special Issue on formal methods for GALS design (2005).