Journal of Circuits, Systems, and Computers

POLYCHRONY* FOR SYSTEM DESIGN

PAUL LE GUERNIC
JEAN-PIERRE TALPIN
JEAN-CHRISTOPHE LE LANN†

INRIA *project* ESPRESSO‡
IRISA, *Campus de Beaulieu, 35042 Rennes, France*

Rising complexities and performances of integrated circuits and systems, shortening time-to-market demands for electronic equipments, growing installed bases of intellectual property (IP), requirements for adapting existing IP blocks with new services, all stress high-level design as a prominent research topics and call for the development of appropriate methodological solutions. In this aim, system design based on the so-called "synchronous hypothesis" consists of abstracting the non-functional implementation details of a system and lets one benefit from a focused reasoning on the logics behind the instants at which the system functionalities should be secured. With this point of view, synchronous design models and languages provide intuitive (ontological) models for integrated circuits. This affinity explains the ease of generating synchronous circuits and verify their functionalities using compilers and related tools that implement this approach. In the relational mathematical model behind the design language SIGNAL, this affinity goes beyond the domain of purely synchronous circuits, and embraces the context of complex architectures consisting of synchronous circuits and desynchronization protocols: globally asynchronous and locally synchronous architectures (GALS). The unique features of the relational model behind SIGNAL are to provide the notion of *polychrony*: the capability to describe circuits and systems with several clocks; and to support *refinement*: the ability to assist and support system design from the early stages of requirement specification, to the later stages of synthesis and deployment. The SIGNAL model provides a design methodology that forms a continuum from synchrony to asynchrony, from specification to implementation, from abstraction to concretization, from interfaces to implementations. SIGNAL gives the opportunity to seamlessly model circuits and devices at multiple levels of abstractions, by implementing mechanisms found in many hardware simulators, while reasoning within a simple and formally defined mathematical model. In the same manner, the flexibility inherent to the abstract notion of signal handled in the synchronous-desynchronized design model of SIGNAL invites and favors the design of correct by construction systems by means of well-defined transformations of system specifications (morphisms) that preserve the intended semantics and stated properties of the architecture under design. The aim of the present article is to review and summarize these formal, correct-by-construction, design transformations. Most of them are implemented in the POLYCHRONY tool-set, allowing for mixed bottom-up and top-down design of embedded hardware-software system using the SIGNAL design language.

*From the Greek *"poly chronos"* to mean multiple clocks
†Present affiliation: Thomson Multimedia R&D France
‡URL `http://www.irisa.fr/espresso`

1 Introduction

High-level design and modeling of hardware and embedded hardware/software systems have gained prominence in the face of rising technological complexity, increasing performance requirements and shortening time to market demands for electronic equipments. Today, the installed base of intellectual property (IP) further stresses the requirements for adapting existing IPs with new services within complex integrated architectures, calling for appropriate mathematical models and methodological approaches to integrate systems.

Over the past decade, numerous programming models, languages, tools and frameworks have been proposed to design, simulate and validate heterogeneous systems within abstract and rigorously defined mathematical models. Formal design frameworks provide well-defined mathematical models that yield a rigorous methodological support for the trusted design, automatic validation, and systematic test-case generation of systems.

However, they are usually not amenable to direct engineering use and not seem to satisfy the present industrial demand. As a matter of fact, the attention of the EDA industry tends to shift to modeling frameworks based on general-purpose programming language variants, in response to a growing industry demand for higher abstraction-levels in the system design process and an attempt to fill the so-called *industrial productivity gap*.

Whereas abstract frameworks are ways to unambiguously model the essence of hardware software systems, to help understand the design, implement formal correctness proofs and predict performances and other metrics; general-purpose languages facilitate programming, encourage reuse and gain from the popularity of C-like and Java-based languages.

At present, a possibility of widening existing divergences between formal methods and industrial practices is perceivable. It seems that any useful methodology cannot avoid the industrial trend of using emerging programming languages. This contrasted picture calls for an effort toward the convergence between the theory of formal methods and the industrial practice and trends in system-design.

The present article aims at this convergence by considering the formal modeling framework POLYCHRONY (as one of many formal design frameworks proposed over the past decade). It invites to a reflection on the implementation of present industrial practices and trends in such a framework by considering the required definition of proper methodological approaches and its implementation by means of effective program analysis and transformation techniques.

1.1 A polychronous design model

Despite the overwhelming advance of Electronic Design Automation (EDA), existing techniques and tools merely provide *ad-hoc* solutions to challenging issues. The pressing demand for design tools has sometimes hidden the need to lay mathematical

foundations below design languages. Many illustrating examples can be found : one of the most striking examples is the variety of very different formal semantics found in hardware description languages (HDLs), even between the most widely used ones, VHDL or Verilog. Even thought these design languages benefit from decades of programming practice, they still give rise to some diverging interpretations of their semantics.

The need for higher abstraction-levels and the rise of stronger market constraints in the EDA industry now make the need for unambiguous design models more obvious. For instance, HDL simulators now need to accelerate the execution through cycle-based and compiled code strategies to support the simulation of chips that score millions of gates.

This challenge requires models and methods to reason on the translation of concurrent processes into purely sequential ones, to implement high-level, non-trivial optimizations such as process folding. Another example is sub-micronic technology, which requires designers to reason about logic optimizations in an entirely different way (wires can now be viewed as connected by gates). Even worth mentioning is that these examples even exhibit apparent contradictions in terms of modeling requirements : the first tends to abstract away from low-level details, while the second states that logic and time are indeed dependent.

In the context of such a contrasted picture, co-design becomes the central activity, instead of just being the interface between distinct hardware and software activities. In the co-design activity, communications between processes is central, interoperability between heterogeneous models and languages is central. All the leverages in system design call for the need to clearly identify Models of Computation (MoC) and develop adequate techniques based on formal computational models and design methods.

In this aim, system design based on the so-called "synchronous hypothesis" has focused the attention of many academic and industrial actors of the EDA community. The synchronous paradigm consists of abstracting the non-functional implementation details of a system and lets one benefit from a focused reasoning on the logics behind the instants at which the system functionalities should be secured. With this point of view, synchronous design models and languages provide intuitive (ontological) models for integrated circuits. This affinity explains the ease of generating synchronous circuits and verify their functionalities using compilers and related tools that implement this approach.

In the relational mathematical model behind the design language SIGNAL, this affinity goes beyond the domain of purely synchronous circuits and embraces the context of complex architectures consisting of synchronous circuits and desynchronization protocols: globally asynchronous and locally synchronous architectures (GALS).

This unique feature is obtained thanks to the fundamental notion of *polychrony*: the capability to describe systems in which components obey to multiple clock rates. It provides a mathematical foundation to a notion of *refinement*: the ability

to model a system from the early stages of its requirement specifications (relations, properties) to the late stages of its synthesis and deployment (functions, automata).

The notion of polychrony goes beyond the usual scope of a programming language, allowing for specifications and properties to be described. As a result, the SIGNAL design methodology draws a continuum from synchrony to desynchronization, from specification to implementation, from abstraction to concretization, from interfaces to implementations. SIGNAL gives the opportunity to seamlessly model circuits and devices at multiple levels of abstraction while reasoning within a simple and formally defined mathematical model.

The inherent flexibility of the abstract notion of signal handled in the synchronous/desynchronized design model of SIGNAL invites and favors the design of correct-by-construction systems by means of well-defined transformations of system specifications (morphisms) that preserve the intended semantics and stated properties of the architecture under design.

Plan In the present article, we propose to examine the mathematical concept of polychrony and define the corresponding relational design model. This model yields the definition of the POLYCHRONY synchronous programming environment. Based on this implementation, we develop top-down (refinement-based) and bottom-up (component-based) methodological approaches to system design. We put this method to work for modeling and synthesizing hardware devices, thereby demonstrating the effectiveness and flexibility of our model and method.

The outline of our presentation starts in section 2 with the exposition of a rationale on the model of SIGNAL. It starts with a highlight on the peculiar design choices of the model, and an informal tutorial presentation of its syntax with key examples. It continues with a formal specification of the underlying mathematical model (section 3). Section 4 assesses the appropriateness of the SIGNAL model for designing complex GALS architectures. This assessment includes a formal presentation of traditional properties as reactivity or determinism, and more specific formal properties such as endochrony, flow-invariance, latency-insensitivity.

Section 5 puts the expression capabilities of our mathematically well-founded model to work by spelling out its methodological consequences. We define both a top-down design engineering method, based on the classical notion of refinement (section 6.1) and then a bottom-up design method, based on the principles of component-based engineering (section 6.2). The introduction of these notions is exemplified by studying the case of the progressive design of an even-parity checker. Section 5.4 presents the static resolution and dynamic model-checking techniques that come with the POLYCHRONY environment to check that the refinement of a design is not regressive and that the integration of components verify the intended functional requirements of the system.

Section 7 probes the usefulness of SIGNAL for engineering circuits further by describing optimization techniques which greatly improve simulation by making use of symbolic transformation methods implemented in the SIGNAL compiler. These

transformations consists of the desynchronization of elementary synchronous components, that, once recombined, allow to produce optimized control paths and generate optimized simulation (C) code. At last, section 8 recapitulates the results implemented in the POLYCHRONY environment (see the ESPRESSO project web-pages (`http://www.irisa.fr/espresso` for more detail) and makes some conclusive remarks.

2 Polychrony

The synchronous approach provides an unambiguous notion of time : time is seen as a chronology, instead of a chronometry, as in traditional HDLs. However, it is not always clear up to what extent synchronous languages are suitable tools to address the various levels of a design; how an implementation can be derived from a specification, and how the latter can proved correct against the former.

2.1 Rationale for a polychronous model

Synchronous design and synchronous languages share a common point of view : the propagation time of signals along combinational paths is always considered as much smaller than the sampled time interval between two observations. Because of the simplicity and popularity of this concept, the synchronous approach is usually subject to the *a priori* that it only applies for the design of purely synchronous circuits.

Polychrony and over-sampling This argument does not (at least) hold in the case of SIGNAL. The polychronous mathematical model associates individual clocks (set of instants, also called time sets) to signals.

Over-sampling (i.e. adding an arbitrary yet inductive number of instants between two clock cycles) enables the specification of constraints between inputs and outputs in such a way that no further input may occur as long as the given constraints are not meet by the (intermediate) calculations of the output.

Over-sampling is an example of flow-preserving transformation or refinement which consists of specifying the internal reactions of a given system component as fast enough to stabilize until new values occur on input and results occur along outputs (in particular, over-sampling does not loose any input value).

The abstract notion of absence (of values) between two occurrences of useful values along a signal is flexible enough to enable reasoning on the successive refinements and transformations of the description of a system. Typically, absence allows to model asynchronous loops in hardware design (e.g. RS latches).

Polychrony and flow-graphs A control-data flow graph (CDFG) is a commonly used representation to capture sequentiality, conditional branching and control loop constructs (in its control-flow portion of the CDFG), and operations on data (in its

data-flow portion of the CDFG). In our model, a clock is associated to every signal; this notion of clock (or time set) generalizes to actions as well as to sequences of actions, providing a uniform way to support data and control transformations on what we name a conditional flow graph.

Abstraction and refinement The notion of over-sampling clearly renders the multiple levels of abstraction and the multiple granularities of time that prevail at the successive architecture levels of a circuit or system. Over-sampling is also very useful to reproduce typical synchronization patterns encountered at system-level between complex components, as well as typical simulation mechanisms that emulate flows of data between loosely coupled domains.

As a result, polychrony exhibits a continuum between the dual notions of abstraction and refinement which is central to decompose, understand and integrate the design of a system at its successive architectural levels of its components. SIGNAL allows the description of properties and behaviors, of interfaces and implementations, of abstractions and refinements within the same formal design model.

Furthermore, the notion of polychrony naturally yields a partial order relation of implementation: a behavior implements a property (or another, more general, behavior), a component implements an interface, a refinement implements an abstraction. This implementation relation can easily be built by means of formal simulation relations and related model-checking and static-checking techniques.

Polychrony and state machines In SIGNAL, model-checking is preferably implemented by mean of the resolution of polynomial dynamical system of equations expressed over the ring $\mathbb{Z}/3\mathbb{Z}$ (using ternary decision diagrams), where absence is interpreted as 0. The main advantage of this representation is that, making explicit use of operations on ideals and variety of $\mathbb{Z}/3\mathbb{Z}$ avoids the enumeration the state-space of a given specification, and additionally enables control-synthesis.

Model-checking SIGNAL specifications could also be implemented using more conventional data-structures, such as boolean decision diagrams or automata. However, experiments shows that such representations significantly increase the number of variables.

In SIGNAL, static-checking is implemented by means of the inference and resolution of invariants between signals, which describe their synchronization, serialization and causality relations: the *clock calculus*. Synchronization and causality relations play a central role for the compilation, optimization and transformation of programs. They are of most interest to statically check the implementation relation: a component implements an interface if the synchronization and causal relations of the interface are implied by those of the component.

2.2 An informal introduction to core-SIGNAL

In SIGNAL, a process P consists of the simultaneous composition of equations over signals. A signal $x \in \mathcal{X}$ describes a possibly infinite flow of timed values (we write $\boldsymbol{x}$ for a sequence of names and assume an infinite set of signal names $\mathcal{X}$). We first introduce the abstract syntax used in this paper and then give some guidelines to read the examples written in SIGNAL.

Abstract syntax An equation $\boldsymbol{x} = f\boldsymbol{y}$ describes a relation between a sequence of operands $\boldsymbol{y}$ and a sequence of results $\boldsymbol{x}$ by a process $f \in F$. We write $v, w \in \mathcal{V}$ for a value and tt and ff for the boolean constants true and false.

The synchronous composition $P \parallel Q$ of the processes P and Q consists of simultaneously considering a solution of the equations in P and Q at any time. The abstract syntax of a process P in core-SIGNAL is defined by the inductive grammar:

$$\begin{array}{rcll} P & ::= & \boldsymbol{x} = f\,\boldsymbol{y} & \text{(equation)} \\ & | & P \parallel Q & \text{(composition)} \\ & | & P \,/\, x & \text{(restriction)} \end{array}$$

SIGNAL requires three primitive processes: pre (to reference the previous value of a signal in time), when (to sample a signal) and default (to deterministically merge two signals). It also requires the following primitive boolean functions: negation not, equality eq and identity id. pre cab be considered as a register whilst when and default are related to control.

$$f \in F \supseteq \{\, \mathsf{pre}\, v \mid v \in \mathcal{V}\} \cup \{\, \mathsf{when}\,, \mathsf{default}\,, \mathsf{not}\,, \mathsf{eq}\,, \mathsf{id}\,, \ldots\}$$

The equation $x = \mathsf{pre}\, v\, y$ initially defines x by the value v and then by the previous value of y in time. It requires x and y to be synchronous (i.e. to be present at the same time).

$$\begin{array}{rl} y : & (t_1, v_1)\,(t_2, v_2)\,(t_3, v_3)\ldots \\ \mathsf{pre}\ v\, y : & (t_1, v)\ \ (t_2, v_1)\,(t_3, v_2)\ldots \end{array}$$

The equation $x = y$ when z defines x by y when z is true.

$$\begin{array}{rl} y : & (t_1, v_1)\,(t_2, v_2)\ (t_3, v_3)\qquad \ldots \\ z : & \qquad (t_2, tt)\ \ (t_3, ff)\ (t_4, tt)\ldots \\ y\ \mathsf{when}\ z : & \qquad (t_2, v_2)\qquad\qquad \ldots \end{array}$$

The equation $x = y$ default z defines x by y when y is present and by z otherwise.

$$\begin{array}{rl} y : & \qquad (t_2, v_2)\,(t_3, v_3)\ \ldots \\ z : & (t_1, v_1)\qquad (t_3, w_3)\ldots \\ y\ \mathsf{default}\ z : & (t_1, v_1)\,(t_2, v_2)\,(t_3, v_3)\ \ldots \end{array}$$

Concrete syntax of SIGNAL If `P` (resp. `exp` or `f(x1,...,xn)`) is a concrete SIGNAL process, we write P (resp. *exp* or $f(\boldsymbol{y})$) to denote its abstract syntax.

- `(x1,...,xn) := f(y1,...,yn)` is the concrete syntax for an equation $\boldsymbol{x} = f(\boldsymbol{y})$;
- The expressions `not c`, `x when c`, `x default y` and `P | Q` have identical concrete and abstract syntax. Delay is written `x:=y$1 init v` in the concrete syntax and $x = \mathsf{pre}\, v\, y$ in the abstract syntax;
- The declaration `P where t x` defines the type `t` of the signal `x` and becomes $P \,/\, x$ in the abstract syntax. The type `event` is the subtype $\{\sharp\} \subset \mathbb{B}$.
- The equations `c := (exp1 = exp2)` and `x := y` in the concrete syntax correspond to the statements $c = \mathsf{eq}\,(exp1, exp2)$ and $y = \mathsf{id}\, x$ in the abstract syntax;
- The equation `x := v` (for `v` a constant) is equivalent to `x := x$1 init v` and `x := when c` to `x := true when c`.

Further notations of the concrete syntax can similarly be expanded into primitive expressions. Examples are `x ^= y`, to mean that x and y are synchronous; `x ^+ y`, to mean that either `x` or `y` are present.

We exemplify the equational design model of SIGNAL by considering examples (written in the concrete syntax) that unveil the key features of its primitive operators and help understanding the peculiarities of the mathematical model to be presented next.

Example 1 (counter). To start with, let us consider putting the three primitive operators together by designing a rudimentary increasing counter that can be reset to 0. The process Count accepts an input reset signal and delivers the integer output signal val. The local variable counter is initialized to 0 and stores the previous value of the signal val (equation `counter := val$1 init 0`). When an input reset occurs, the signal val is reset to 0 (expression `(0 when reset)`). Otherwise, the signal val takes an increment of the variable counter (expression `(counter + 1)`).

```
process Count = (? event reset ! integer val)
  (|  counter := val$1 init 0
   |  val     := (0 when reset) default (counter + 1)
   |) where integer counter;
end;
```

Count events	·	·	·	·	·	·	·	·	·	·	·	·	·	·
reset		$\sharp$					$\sharp$				$\sharp$	$\sharp$		
val	1	0	1	2	3	4	0	1	2	3	0	0	1	2
counter	0	1	0	1	2	3	4	0	1	2	3	0	0	1

Notice that the activity of the process Count is governed by the clock of its output val which differs from that of its input reset. Count is a polychronous process. If the signal val is solicited by the environment, then either reset is absent and Count increments val, or reset is present and Count sets `val` to 0. Hence, the process Count is reactive and deterministic.

Example 2 (sampler). It can be used in different contexts in order to provide the required behaviors. For instance, in the process Sampler, the process Count is wrapped by additional input and output signals in order to implement a counter modulo N (an integer constant parameter) and can still be reset to 0 upon receipt of the reset event. Had we written `val := Count(alarm)` instead of `val := Count(reset default alarm)`, we would have defined a modulo function instead.

```
process Sampler = {integer N} (? event reset, tick ! integer val; event alarm)
   (|  val   := Count(reset default alarm)
    |  mod   := (val = N-1)$1 init true
    |  alarm := when mod
    |  val   ^= reset ^+ tick
    |) where boolean mod;
end;
```

observable events	·	·	·	·	·	·	·	·	·	·	·	·	·	·
tick	↯		↯	↯	↯	↯	↯	↯	↯	↯	↯		↯	↯
reset		↯									↯	↯		
alarm							↯							
val	1	0	1	2	3	4	0	1	2	3	0	0	1	2
counter	0	1	0	1	2	3	4	0	1	2	3	0	0	1

The rate of execution is now governed by two input signals tick and reset. An alarm signal is generated each time the count val is reset to 0 (i.e. if the previous output of Count was $N-1$, expression `mod := (val = N-1)$1 init true`).

In this example, we also introduce synchronization relations. To ensure a correct synchronization of the signal val with the reset and tick events, we need to specify the relation `val ^= reset ^+ tick`, which means that val is present (and activates the counter) iff either reset or tick is present.

Example 3 (clock). We may use the sampler to modularly design a clock (with a reset capability) by composing four instances of the parameterized Sampler process to count seconds, minutes and hours given an input base signal and the constant number N of bases within a second.

```
process Clock = {integer N} (? event reset, base ! integer hours, mins, secs)
   (| (units, s) := Sampler {N}  (reset, base)
    | (secs, m)  := Sampler {60} (reset, s)
    | (mins, h)  := Sampler {60} (reset, m)
    | (hours, d) := Sampler {24} (reset, h)
    |) where integer units; event s, m, h, d;
end;
```

Example 4 (over-sampling). We now consider more advanced modeling techniques by considering the design of a true piece of hardware and exercise elementary methodological notions by considering the refinement of this design and its implementation, making use of the unique feature of over-sampling, that characterizes the SIGNAL model.

In SIGNAL, one may want to design a process making internal iterative (and finite) computations (just as for counting seconds), but sometimes, one would appreciate to just have to care about the result of the iteration and not to have to explicitly activate that process to perform an iteration. It is not necessary to do so in SIGNAL, by contrast with related synchronous languages.

As an example, consider the following expression which checks a particular byte-level property (sub-process `IsEven`) of an integer signal num. At this level of design, one just expects `IsEven` to return an event when the property is satisfied (hence, the boolean output of IsEven is down-sampled, using when, to the instants at which it holds the value true).

$$\texttt{even} := \texttt{true when IsEven}(\texttt{num})$$

Now, suppose that the property implemented by the process IsEven consists of checking teh condition that the number of bits of the input is an even number. Although the computation of this property can be done in a bounded number of iterations, it requires the activation of a sub-clock every time an input count is passed to the process `IsEven`. To implement it in SIGNAL, one uses an internal, variable over-sampling.

```
process IsEven = (? integer num ! boolean parity)
   (|  num    ^= start
    |  start  := when (done$1 init true)
    |  parity := flip when done
    |  done   := mask=0
    |  (| curmask := rshift (mask$1 init 0)
        | mask    := num default curmask
        |) where integer curmask
           end
    |  (| tick      := when (xand (mask, 1) = 1)
        | resetFlip := (xand (num, 1) = 0)
        | flip      := resetFlip default ((not flop) when tick) cell ^done
        | flop      := flip$1 init true
        |) where event tick;
                  boolean resetFlip, flop
           end
    |) where event start;
             boolean done, flip;
             integer mask;
             function rshift = (? integer x ! integer y);
             function xand = (? integer x, y ! integer z);
end;
```

The process IsEven is initially in standby. It starts when an input num value is

present (expression `num ^= start`). It returns in standby after termination (expression `start := when done$1 init true`). The process terminates when the internal variable mask reaches 0, meaning that there are no remaining bits to count (expression `done := mask=0`).

The variable mask is calculated every time the process is active and until the termination condition holds (meaning that a fix-point is reached). It is initialized to the input signal num when the process starts. Then, every time the process is active, it is shifted right `curmask := rshift (mask$1 init 0)`.

A tick occurs when a bit in the mask is set. This is determined by performing a logical-and between the mask and 1. If a tick occurs, a boolean flip signal commutes. The variable flop holds its previous value and is initially true. The operator cell allows to write a register cell at the clock-rate of the sub-expression (not flop) when tick and read it at the (different) clock-rate of the signal done. When the process stops, the parity is true iff that signal is *true*. This example makes use of external function declarations (rshift and xand).

observable events	·			·	·			·	·	·		·
num	5				4				0	3		
parity				*tt*				*ff*	*tt*			*tt*
additional internal events	ı	·	·	ı	ı	·	·	ı	ı	ı	·	ı
done	*ff*	*ff*	*ff*	*tt*	*ff*	*ff*	*ff*	*tt*	*tt*	*ff*	*ff*	*tt*
mask	5	2	1	0	4	2	1	0	0	3	1	0
tick	*tt*		*tt*				*tt*			*tt*	*tt*	
flip	*ff*	*ff*	*tt*	*tt*	*tt*	*tt*	*ff*	*ff*	*tt*	*ff*	*tt*	*tt*

3 A model for polychronous systems

Starting from the model of tagged signals of Lee and Sangiovani [19], we define a tagged model of polychronous signals. This yields a structure of polychronous processes that is used to give the formal trace semantics of core-SIGNAL. We then assess the generality and expressive power of the formal semantics and design model of core-SIGNAL by spelling out its algebraic properties.

3.1 A tagged model of polychronous processes

We consider a set of boolean and integer *values* $v \in \mathcal{V} = \mathbb{B} + \mathbb{Z}$ to represent the operands and results of a computation. A tag t, used to denote synchronization, is an element of a set $\mathbb{T}$ equipped with a *partial order* relation $\leq$; the partial order represents causality. A *chain* $C \in \mathbb{T}$ is a totally ordered subset of $\mathbb{T}$.

Definition 1 (partial-order of tags). The partially ordered set $(\mathcal{T}, \leq)$ of a given process is a subset $\mathcal{T} \subset \mathbb{T}$ that satisfies the following properties:

$\mathcal{T}$ is countable
$\mathcal{T}$ has a lower bound 0 for $\leq$
the partial-order $\leq$ on $\mathcal{T}$ is well-founded

Let $\mathcal{C}$ be the set of chains C in $\mathcal{T}$. Then, for a tag $t \in C$, we write $\min(C)$, $\max(C)$ and $\mathrm{pred}_C(t)$ for the minimum, maximum and immediate predecessor of t in C.

Definition 2 (event, signal and behavior). An *event* $e \in \mathcal{E} = \mathcal{T} \times \mathcal{V}$ is a relation between a tag and a value. A *signal* $s \in \mathcal{S} = \mathcal{T} \rightharpoonup \mathcal{V}$ is a partial function defined on a *chain* of tags to a set of values. We write $\mathrm{tags}(s)$ to denote the domain of s. A *behavior* $b \in \mathcal{B} = \mathcal{X} \rightharpoonup \mathcal{S}$ is a partial function from signal names $x \in \mathcal{X}$ to signals $s \in \mathcal{S}$. We write $\mathrm{vars}(b)$ to denote the domain of b and $\mathrm{tags}(b) = \cup_{x \in \mathrm{vars}(b)} \mathrm{tags}(b(x))$ to denote its tags. Hence, the informal sentence "x is present (at t in b)" can be formally defined by $t \in \mathrm{tags}(b(x))$.

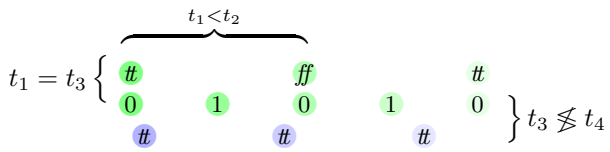


Figure 1: A behavior b: from signal names to partially ordered tags and values

We write $b|_X$ for the projection of a behavior b on a set $X \subset \mathcal{X}$ of names (i.e. $\mathrm{vars}(b|_X) = X$ and $\forall x \in X, b|_X(x) = b(x)$) and $b_{/X}$ for the projection of b on the complementary of X in $\mathrm{vars}(b)$ (i.e. $b_{/X} = b|_{\mathrm{vars}(b) \setminus X}$). We write λ to denote the empty signal and $\mathsf{0}|_X = \{(x, \lambda) \,|\, x \in X\}$ to associate $X \subset \mathcal{X}$ to the empty signal.

Definition 3 (process). A *process* $p \in \mathcal{P} = \mathcal{P}(\mathcal{B})$ is a set of behaviors that have the same domain X (we write it $\mathrm{vars}(p)$). The synchronous composition $p \,|\, q$ of two processes p and q is defined by the set of behaviors that extend a behavior $b \in p$ by the restriction $c_{/\mathrm{vars}(p)}$ of a behavior $c \in q$ provided that the projections of b and c on $\mathrm{vars}(p) \cap \mathrm{vars}(q)$ are equal[§]

$$p \,|\, q = \left\{ b \uplus c_{/\mathrm{vars}(p)} \,|\, (b, c) \in p \times q,\ b|_{\mathrm{vars}(p) \cap \mathrm{vars}(q)} = c|_{\mathrm{vars}(p) \cap \mathrm{vars}(q)} \right\}$$

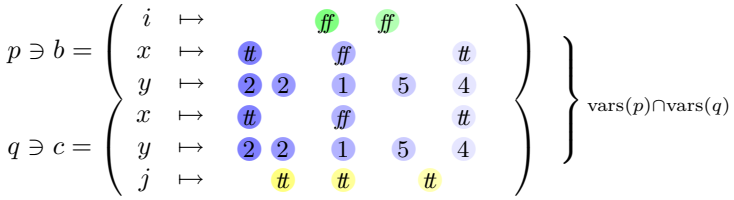


Figure 2: Synchronous composition $p \,|\, q$: matching behaviors along common signals

[§]or, equivalently, $b/\mathrm{vars}(q) \uplus b|_{\mathrm{vars}(p) \cap \mathrm{vars}(q)} \uplus c/\mathrm{vars}(p)$ since b and c coincide on $\mathrm{vars}(p) \cap \mathrm{vars}(q)$

Notes Notice that a process p consists of an inductive number of behaviors, and that a behavior consists of a finite number of tag chains. Each chain is infinite and countable. Hence, the set of tags in a process is at most countable. In the present model, a signal is a partial function from a set of tags $\mathcal{T}$ to a set of values $\mathcal{V}$. An alternative approach would be to define it as a total function on $\mathcal{T}$ to $\mathcal{V}$ completed by $\perp$. In [19], a more general definition of process is given by considering an unordered set of tags and a set of value completed with a special mark $\perp$ to denote the absence of a value at a given time tag. This generality is not required to model synchronous processes. Synchronous structures [21] also constitute a more general model, where the so-called imaginary signals (i.e. signals in which tags are not totally ordered) can also be modeled.

3.2 Scalable design

A key concept for defining systems and reusable components in a smooth design process is the concept of scalable observation. For instance, when defining the instruction set of a component, the various micro steps of the execution of an instruction don't have to be known, and the instruction is seen as a process that takes arguments and (simultaneously, or at the next step) does an atomic action; the designer of the circuit will actually implement a finer clocked process. A single virtual circuit can be implemented using various internal architectures and various clock frequencies. A technical support for allowing time scalability is given in our model by the so called stretch-closure property, strongly related to stuttering invariance in models with silent events.

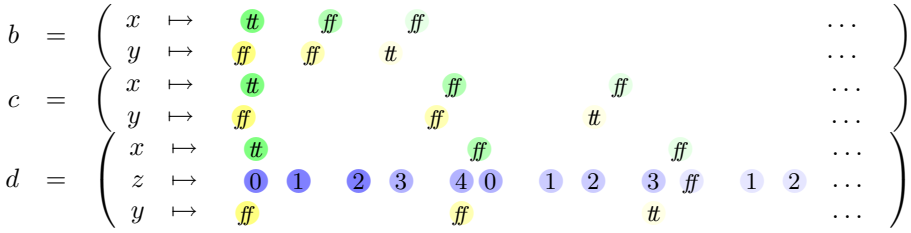


Figure 3: Stretching b allows for a scalable c and refinement d

The intuition behind this relation is to consider a signal as an elastic with (ordered) marks on it (the tags). If we stretch the elastic, the marks remain in the same order, but we may now add more marks between two stretched marks. If we unstretch the elastic, all marks will be closer to one another and some of them not distinguishable, but they will still remain in the same order. The same holds for a set of elastics: a behavior. If we equally stretch each elastic, the partial order between each mark will remain the same.

Definition 4 (stretching). A behavior c is a stretching of b, written $b \leq c$, iff $\mathrm{vars}(b) = \mathrm{vars}(c)$ and there exists a bijection $f : \mathcal{T} \to \mathcal{T}$ that
- is order-preserving, i.e. $\forall t, u \in \mathcal{T},\ t < u \Leftrightarrow f(t) < f(u)$
- is increasing, i.e. $\forall C \in \mathcal{C},\ \forall t \in C,\ t \leq f(t)$
- satisfies $\mathrm{tags}(c(x)) = f(\mathrm{tags}(b(x)))$ for all $x \in \mathrm{vars}(b)$
- satisfies $b(x)(t) = c(x)(f(t))$ for all $x \in \mathrm{vars}(b)$ and all $t \in \mathrm{tags}(b(x))$

The relation of stretching $\leq$ is a partial-order relation on behaviors. It gives rise to an equivalence relation between behaviors. Both relations are extended to processes.

Definition 5 (stretch-equivalence). The behaviors b and c are stretch-equivalent, written $b \lesseqgtr c$, iff there exists a behavior d s.t. $d \leq b$ and $d \leq c$.

It is appealing to consider the relation between processes and the partial order of stretching. In particular, it is interesting to consider the class of processes which contain all possible stretches of a given behavior.

Definition 6 (stretch-closure). A process p is stretching-closed iff for all $b \in p$ and all $c \in \mathcal{B}$, $c \lesseqgtr b \Rightarrow c \in p$.

A non-empty, stretching-closed process p admits a set of strict behaviors, written $(p)_{\lesseqgtr}$, s.t. $(p)_{\lesseqgtr} \subset p$ (for all $b \in p$, there exists a unique $c \in (p)_{\lesseqgtr}$ s.t. $c \lesseqgtr b$).

*3.3 Denotation of core-*SIGNAL

The denotation $[\![P]\!]$ of a process P consists of the largest set of behaviors accepted by P. It is defined by induction on the structure of P. For each equation, $\boldsymbol{x} = f\boldsymbol{y}$ and by induction hypothesis, the function $[\![]\!]$ defines the relation between the signals involved in an equation by considering the chain of tags that supports it.

Denotation function $[\![p]\!]$ The equation $x = \mathsf{pre}\, v\, y$ initially defines x by the value v and then by the previous value of y in time. Suppose that the value of the input signal y is w at a given tag t. The output signal x loads the value v and the process evolves as $x = \mathsf{pre}\, w\, y$, storing the value w of y. The meaning of the equation $x = \mathsf{pre}\, v\, y$ is to define the signals x and y along that same chain of tags $C \in \mathcal{C}$ and define the value of x at a given tag t by the value of y at the immediate predecessor of t in C. Notice that this requires x and y to be synchronous (i.e. to share the same tags C).

$$[\![x = \mathsf{pre}\, v\, y]\!] = \{\mathsf{0}|_{x,y}\} \cup \left\{ b \in \mathcal{B}|_{x,y} \;\middle|\; \begin{array}{l} \mathrm{tags}(b(x)) = \mathrm{tags}(b(y)) = C \in \mathcal{C} \setminus \emptyset,\ b(x)(\min(C)) = v \\ \forall t \in C \setminus \min(C),\ b(x)(t) = b(y)(\mathrm{pred}_C(t)) \end{array} \right\}$$

The equation $x = y \,\mathsf{when}\, z$ defines x by y when z is true. Let us consider the tag t at which this happens. There are four cases to consider:

- if $t \notin \text{tags}(y)$ then $t \notin \text{tags}(x)$
- if $t \notin \text{tags}(z)$ then $t \notin \text{tags}(x)$
- if $t \in \text{tags}(y)$ and $t \in \text{tags}(z)$ and $z(t) = ff$ then $t \notin \text{tags}(x)$
- if $t \in \text{tags}(y)$ and $t \in \text{tags}(z)$ and $z(t) = tt$ then $t \in \text{tags}(x)$ and $x(t) = y(t)$

$$[\![x = y \,\mathsf{when}\, z]\!] = \left\{ b \in \mathcal{B}|_{x,y,z} \;\middle|\; \begin{array}{l} \text{tags}(b(x)) = \{t \in \text{tags}(b(y)) \cap \text{tags}(b(z)) \,|\, b(z)(t) = tt\} \\ \forall t \in \text{tags}(b(x)),\ b(x)(t) = b(y)(t) \end{array} \right\}$$

The equation $x = y \,\mathsf{default}\, z$ defines x by y when y is present and by z otherwise. At a given tag t, there are three cases to consider:

- if $t \in \text{tags}(y)$ then $t \in \text{tags}(x)$ and $x(t) = y(t)$
- if $t \notin \text{tags}(y)$ and $t \in \text{tags}(z)$ then $t \in \text{tags}(x)$ and $x(t) = z(t)$
- if $t \notin \text{tags}(y)$ and $t \notin \text{tags}(z)$ then $t \notin \text{tags}(x)$

$$[\![x = y \,\mathsf{default}\, z]\!] = \left\{ b \in \mathcal{B}|_{x,y,z} \;\middle|\; \begin{array}{l} \text{tags}(b(y)) \cup \text{tags}(b(y)) = \text{tags}(b(x)) = C \in \mathcal{C} \\ \forall t \in C,\ b(x)(t) = \left\{ \begin{array}{ll} b(y)(t), & t \in \text{tags}(b(y)) \\ b(z)(t), & t \notin \text{tags}(b(y)) \end{array} \right. \end{array} \right\}$$

$$[\![x = \mathsf{pre}\, tt\, y]\!] \ni \left(\begin{array}{lllllll} y & \mapsto & ff & tt & ff & tt & \cdots \\ x & \mapsto & tt & ff & tt & ff & tt \quad \cdots \end{array} \right)$$

$$[\![x = y \,\mathsf{default}\, z]\!] \ni \left(\begin{array}{lllllll} y & \mapsto & ff & & ff & tt & \cdots \\ z & \mapsto & tt & ff & & ff & tt \quad \cdots \\ x & \mapsto & ff & ff & ff & tt & tt \quad \cdots \end{array} \right)$$

$$[\![x = y \,\mathsf{when}\, z]\!] \ni \left(\begin{array}{lllllll} y & \mapsto & tt & tt & & & tt \quad \cdots \\ z & \mapsto & ff & & ff & tt & tt \quad \cdots \\ x & \mapsto & & & & & tt \quad \cdots \end{array} \right)$$

Figure 4: Delay, sampling and merge

The meaning of the synchronous composition $P \,|\, Q$ is the synchronous composition $[\![P]\!] \,|\, [\![Q]\!]$ of the denotations $[\![P]\!]$ and $[\![Q]\!]$.

$$[\![P \,|\, Q]\!] = [\![P]\!] \,|\, [\![Q]\!]$$

From the denotation of $[\![P \,|\, Q]\!]$, one easily infers the following property:

Property 1. Composition is commutative and associative

The meaning of restriction P/x (restricting the scope of the signal x to the process P) consists of all behaviors c that are stretch-equivalent to a behavior b/x from $[\![P]\!]$. Notice that the restriction of b over x is not enough to obtain all stretch-equivalent behaviors (see example 5). Therefore, one needs to define the meaning of $[\![P/x]\!]$

as all $c \leq b/x$ for all $b \in [\![P]\!]$. Using the same symbol for syntactic and semantic restriction we define:

$$[\![P/x]\!] = [\![P]\!]/x \text{ and } p/x = \{c \,|\, b \in p \wedge c \leq (b/x)\}$$

Example 5. For instance, let $b = ((x, s_1), (y, s_2))$ with $s_1 = \{(t_1, 1), (t_2, 2), (t_3, 3)\}$ and $s_2 = \{(t_1, \#), (t_3, \#)\}$ and consider the process $p = \{c \geq b\}$. The signal s_2 is stretch equivalent to $s_2' = \{(t_1, \#), (t_2, \#)\} \leq s_2$. Hence, s_2' should be in the restriction of p over x, but it is not contained in $\{c_x \,|\, c \in p\}$.

The denotation of processes defined by the function $[\![P]\!]$ satisfy the requirements for a scalable design. Namely, a process P can be used at different time scales because its denotation is closed for the stretch-equivalence relation.

Property 2. For all SIGNAL processes P, $[\![P]\!]$ is stretch-closed.

3.4 Algebraic properties

We now introduce the algebraic structure of processes in core-SIGNAL. We first consider some remarkable processes: the process $\mathtt{1}$ s.t. $\text{vars}(\mathtt{1}) = \emptyset$ is neutral w.r.t. synchronous composition; the process $\mathtt{0} = \{(x, \lambda)_{x \in X} \,|\, X \subset \mathcal{X}\}$ is absorbent w.r.t. synchronous composition. Let us note $\text{vars}(P)$ (resp. $\text{out}(P)$) for the set of (output) signals in P and define $\text{in}(P) = \text{vars}(P) \setminus \text{out}(P)$ for the input signals of P.

$$\begin{array}{rl} \text{vars}(\boldsymbol{x} = f\boldsymbol{y}) = & \boldsymbol{x} \cup \boldsymbol{y} \\ \text{vars}(P \,|\, Q) = & \text{vars}(P) \cup \text{vars}(Q) \\ \text{vars}(P/x) = & \text{vars}(P) \setminus \{x\} \end{array} \qquad \begin{array}{rl} \text{out}(\boldsymbol{x} = f\boldsymbol{y}) = & \boldsymbol{x} \\ \text{out}(P \,|\, Q) = & \text{out}(P) \cup \text{out}(Q) \\ \text{out}(P/x) = & \text{out}(P) \setminus \{x\} \end{array}$$

A key property regards the containment of local signals. It is central to establish the algebraic structure of polychronous processes. It stipulates that a process P is equivalent to the projection of the composition $P \,|\, Q$ on the signals $\text{vars}(P)$ iff the projection of Q on the signals $\text{vars}(P)$ is contained in the projection of P on $\text{vars}(Q)$. This observation yields the algebraic structure of $[\![P]\!]$.

Property 3. For all processes P and Q s.t. $\text{vars}(P) = A$ and $\text{vars}(Q) = B$,

$$\left|\begin{array}{l} ([\![P]\!] \,|\, [\![Q]\!])|_A \subset [\![P]\!] \\ ([\![P]\!] \,|\, [\![Q]\!])|_B \subset [\![Q]\!] \\ [\![P]\!] = ([\![P]\!] \,|\, [\![Q]\!])|_A \text{ iff } [\![Q]\!]|_{A \cap B} \supset [\![P]\!]|_{A \cap B} \end{array}\right.$$

The syntactic containment properties give rise to the structural equivalence laws between processes of $[\![P]\!]$.

Corollary 1 (laws of composition). For all processes P, Q and R, we have

absorbent:	$[\![P]\!] \,|\, \emptyset = \emptyset$
idempotent:	$[\![P]\!] \,|\, [\![P]\!] = [\![P]\!]$
monotonous:	$[\![P]\!] \subset [\![Q]\!] \Rightarrow [\![P]\!] \,|\, [\![R]\!] \subset [\![Q]\!] \,|\, [\![R]\!]$
neutral:	$[\![P]\!] \,|\, 1 = [\![P]\!]$
	$\text{vars}(P) = \text{vars}(Q) \Rightarrow [\![P]\!] = [\![P]\!] \,|\, [\![Q]\!] \Leftrightarrow [\![P]\!] \subset [\![Q]\!]$

Hence the algebraic structure of $[\![P]\!]$:

Property 4. $([\![P]\!], |, 1)$ is a commutative monoid.

3.5 Flow relations

Stretching defines an equivalence relation that preserves the simultaneousness and the ordering of events within a behavior, i.e. stretched behaviors possess the same synchronization relations. We introduce a weaker relation which discards the actual synchronization relations between signals and allows for comparing behaviors w.r.t. the sequences of values that signals hold. For this purpose we define a *relaxation* relation which allows to individually stretch the signals of a behavior, and the corresponding *flow-equivalence* relations between behaviors.

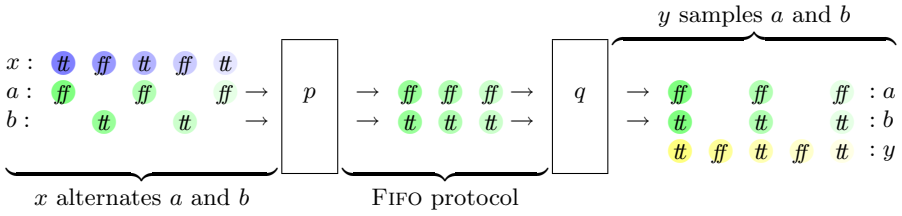


Figure 5: Asynchrony as the relaxation of synchronization relations

Definition 7 (relaxation). A behavior c is a relaxation of b, written $b \sqsubseteq c$, iff $\text{vars}(b) = \text{vars}(c)$ and for all $x \in \text{vars}(b)$, $b_{|x} \leq c_{|x}$.

The relation of relaxation $\sqsubseteq$ is a partial-order relation on behaviors. It gives rise to the flow-equivalence relation between behaviors. Two behaviors are flow-equivalent iff they have the same domain and their signals hold the same values in the same order.

Definition 8 (flow-equivalence). The behaviors b and c are flow-equivalent, written $b \approx c$, iff there exists a behavior d s.t. $d \sqsubseteq b$ and $d \sqsubseteq c$.

Since the equivalence class of a behavior b is a semi-lattice, it admits a strict behavior, written $b_{\approx}$.

3.6 Sampling

Stretching defines an equivalence relation that preserves simultaneousness and ordering of all events, i.e. the stretched behavior is isomorphic to the original one. It mainly addresses the problem of using a component in various contexts. Relaxation defines an equivalence that preserves flows. This relation is useful for modeling the distribution of a process on an asynchronous architecture. The refinement of a system specification consists of transforming its abstract behaviors into more concrete ones that make intermediate computational steps explicit. Conversely, the abstraction of a behavior consists of discarding some intermediate calculations and the related intermediate events. Hence, it is necessary to device an *implementation relation* that takes the notion of time refinement into account for comparing behaviors. For that purpose we define the notions of *sampler* and *sampling* (w.r.t. a sampler). These definitions are the basis for hardware implementation in POLYCHRONY.

Definition 9 (sampler). A signal c *is a sampler of* a signal x iff

$$\begin{array}{|l} \forall t \in \mathrm{tags}(c), c(t) \in \mathbb{B} \\ \mathrm{tags}_{\not t}(x) \subset \ \mathrm{tags}_{\not t}(c) \subset \ \mathrm{tags}(x) \subset \ \mathrm{tags}(c) \end{array}$$

where $\mathrm{tags}_{\not t}(c) = \{t \in \mathrm{tags}(c) \,|\, c(t) = \not t\}$. A sampler c of a signal x is a *clock of* x iff $\mathrm{tags}(x) = \mathrm{tags}_{\not t}(c)$. A clock of x is a *pure clock of* x iff $\mathrm{tags}(c) = \mathrm{tags}_{\not t}(c)$.

Definition 10 (signal sampling). The *sampling of* x *by* c, a sampler of x, consists in "removing" x occurrences from tags on which c is not equal to $\not t$. The result is the signal $y = when_c(x)$ such that

$$\begin{array}{|ll} & \mathrm{tags}(y) = \mathrm{tags}_{\not t}(c) \\ \forall t \in \mathrm{tags}(y), & x(t) = y(t) \end{array}$$

We extend these definitions to behaviors and processes. This requires the definition of a *sampler system*.

Definition 11 (sampler system). Let b a behavior, and A=vars(b). A *sampler system of* b is a function $\kappa : A \to A$ s.t.

$$\begin{array}{|l} \kappa \text{ is acyclic} \\ \forall a \in \mathrm{dom}(\kappa),\ b(\kappa(a)) \text{ is a sampler of } b(a) \end{array}$$

A sampler system κ is a clock system iff for all $a \in \mathrm{dom}(\kappa),\ b(\kappa(a))$ is a clock of $b(a)$. These definitions are extended to processes. A function κ is a sampler system (a clock system) for a process p iff it is a sampler system (a clock system) for every behavior b of p. We can define the notion of sampling.

Definition 12 (sampling). The *sampling of* a behavior b by a sampler system κ is the behavior $b' = \mathcal{S}_\kappa(b)$ s.t. vars(b) = vars(b') and for all $x \in$ vars(b), $b'(x) = \mathcal{S}^* b((x))$ where $\mathcal{S}^*$ is recursively defined by

| if κ is not defined on x then $\mathcal{S}^*(b(x)) = b(x)$
| if κ is defined on x then $S^*(b(x)) = when_{\mathcal{S}^*(b(\kappa(x)))}(x)$

This definition is extended to processes. The *sampling of* (a process) p *by* κ, a sampler system of p, is the process $p' = \{\mathcal{S}_\kappa(b) \,|\, b \in p\}$

Example 6. For instance, $\kappa = \{(x, c)\}$, with c a boolean signal, is a sampler system for the SIGNAL processes `x^=c` and `x^=when c`. It is a clock system for the equation `x^=when c` that is itself a sampling of `x^=c` by κ.

x^= c									
c	tt	tt	ff	tt	ff	ff	tt	tt	ff
x	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9

x^= when c									
c	tt	tt	ff	tt	ff	ff	tt	tt	ff
x	v_1	v_2		v_4			v_7	v_8	

4 Polychronous design properties

In this section, we intend to assess the generality and expressive power of the formal semantics and design model of core-SIGNAL and demonstrate to what extent it encompasses and completes models considered in related works.

Synchronous programming is commonly associated to the notion of reactivity. If a real-time system has to immediately react to some high prioritized events, it is not required to do so for all events. The POLYCHRONY model allows to design reactive systems and "proactive" ones. Determinism is a key property for critical program execution. Nevertheless, specification level, property description, abstractions need non determinism resulting from partial descriptions. In the POLYCHRONY framework, one can design such non deterministic behaviors.

Synchronous programming was initially widely publicized using the idealized model of zero-time computation and instantaneous broadcast communication. Distributed systems do not, however, obey this idealized picture of perfect synchrony: computations and communications take time, interaction topologies evolve during service. As a matter of fact, synchrony and asynchrony are usually perceived as fundamentally different concepts in nature.

Asynchrony (as a model of concurrency) is traditionally relevant for reasoning on distributed algorithms and for modeling non-determinism, failure, mobility. It meets a natural implementation by networked point-to-point communication. Synchrony is largely understood as specific to the design of reactive systems and digital circuits. In this context, timeless logical concurrency and determinism are suitable hypothesis.

On the flip side, a synchronous design hypothesis consists usually of assuming that communications and computations are instantaneous between the successive execution steps of a system. Making this hypothesis is beneficial for design. It allows the designer to focus on the logics of the system, characterized by synchronization and causal relations between events, and abstract away timing issues until a later stage of the design (its deployment on a given architecture).

However, an everyday broader range of software development areas requires reasoning on a combination of synchronous and asynchronous interaction at the different architectural levels of the system under design. Relevant practical examples are co-designed hardware-software architectures, reconfigurable embedded devices, multi-threaded reactive systems components on real-time virtual machines and operating systems, distributed and reactive telecommunication applications on fault-tolerant middle-ware. In summary, every system whose design requires robustness to latency, to threading, to distribution.

4.1 Process properties

Reactivity SIGNAL allows the specification of either reactive or non-reactive yet constrained processes. Following the approach of Berry, we define reactivity w.r.t. a set of signals I (usually, the set of inputs) as the ability of the process to react to each configuration of I in all states.

Definition 13 (configuration). A *configuration on* $I \subset \mathcal{X}$ is a partial function from signal names $x \in I$ to values $v \in \mathcal{V}$.

A process p of variables $X = \text{vars}(p)$ is reactive on a given set $I \subset X$ of inputs iff, for all strict behaviors $b \in p$, for all tags $t \in \text{tags}(b)$, for all configurations s on I (including the empty signal), there exists at least a behavior c that has the same prefix as b up to $\text{pred}(t)$ and such that $c_t = s$. For a signal s (resp. behavior b), we write $s|_{\leq t}$ the prefix of s until the tag $t \in \text{tags}(s)$ i.e. $s|_{\leq t} = \{(t', v) \in s \mid t' \leq t\}$.

Definition 14 (reactive). A process p is reactive (strictly reactive) on $I \subset \text{vars}(p)$ iff, for all $b \in (p)_{\leqslant}$, for all $t \in \text{tags}(b)$, for all configurations (all non empty configurations) s on I there exists $c \in (p)_{\leqslant}$ s.t.:

$$(c)|_{\leq \text{pred}(t)} = (b)|_{\leq \text{pred}(t)}, c_t = s$$

A process that is reactive on a not empty I is (obviously) strictly reactive on I. We say that a process is reactive (strictly reactive) if it is reactive (strictly reactive) on its input signals.

Example 7. The equations `z:=x default y` and `z:=x when y` as well as all basic SIGNAL equations with a single input signal are strictly reactive processes. The

processes `Sampler` and `Clock` are further examples of strictly reactive processes. By contrast, the process `Count` is (only) reactive (see example 1).

On the opposite, the process `z:=x and y` is not reactive at all (in the sense of definition 14). The following program implements an `and` combinator that is "maximally" reactive on `x` and `y`.

```
(|  a := x default (a$1 init false)
 |  b := y default (b$1 init false)
 |  z := a and b
 |) where integer a, b;
```

To obtain a strictly reactive event driven `and` operator, we additionally need to add the synchronization constraint `z ^= x^+ y`. One may also want to extend the operator `and` to have a "minimal" and strictly reactive behavior (i.e. that does nothing as soon as one of its input signals is absent).

```
z := (x when ^y) and (y when ^x)
```

To get a reactive `and` whose output signal `zz` only changes when both input signals are present, one can simply add the equation `zz := z default (zz$1 init false)`. Last, but not least, the processes `IsEven` (example 4) is not strictly reactive. It accepts inputs only in standby (initially) or if its previous computation is terminated.

As a matter of comparison, ESTEREL programs are reactive by design. Depending upon the semantics of its step interpretation, a STATECHART may, or may not, be reactive. Finally, LUSTRE programs are usually not strictly reactive; they are never reactive.

Determinism An automaton is deterministic if, in all of its states, each event corresponds to at most one transition. In our model, a process p s.t. $X = \mathrm{vars}(p)$ is deterministic on a given set $I \subset X$ of signals iff two strict behaviors b and c of p, that have the same projection on I up to a tag t, have the same behaviors (on X) up to t.

Definition 15 (determinism). A process p is deterministic on a (possibly empty) subset $I \subset \mathrm{vars}(p)$ iff, for all $c_1, c_2 \in (p)_{\lessdot}$, for all $t \in \mathrm{tags}(c_1) \cap \mathrm{tags}(c_2)$:

$$(c_1|_I)|_{\leq t} = (c_2|_I)|_{\leq t} \Rightarrow (c_1)|_{\leq t} = (c_2)|_{\leq t}$$

We say that a process is deterministic if it is deterministic on its input signals.

Example 8. The process `x := y$1 init v` is deterministic as well as all other basic SIGNAL equations (on distinct signals). The process `Count` of example 1 is deterministic. By contrast, the process `x := a default x` is not deterministic. When a is absent the value of x is free. Thus x is partially defined by this equation. In SIGNAL, one can specify such a partial definition of x by write `x ::= a` in place

of `x := a default x`. Several partial definitions of a signal x can appear in a program, provided that these definitions are compatible (i.e. the composition `x ::= a | x ::= b` requires a and b to hold the same value when they have the same tag). Fortunately, non-deterministic specifications can be composed to form a deterministic program when some suitable synchronization constraints are added. This is, however, not always possible. For instance, the following process (which makes use of Count, example 1) is intrinsically non deterministic (actually the signal `out` may be every sequence of positive or null integers):

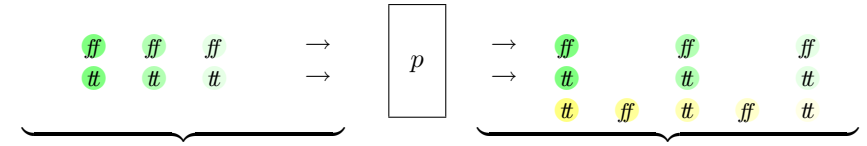
```
(| val := Count(reset) | out := val when request |) where integer val;
```

The reason why it is definitely not deterministic is that, whatever the context is, there is no way to synchronize the internal clock of val and thus the activity rate of this occurrence of Count.

As a matter of comparison, ESTEREL and LUSTRE programs are deterministic by design. Most STATECHART interpretations are usually non deterministic. Signal capability to express non deterministic behaviors makes this language suitable for defining partial specifications, for modeling process or program abstractions, for modeling non deterministic devices, etc.

Endochrony The property of endochrony is a key property for system design, and specially component based system design. It can be seen as the equivalence between the internal (synchronous) and external (asynchronous) observations of a process. A process is said to be endochronous on I iff, given an external (asynchronous) stimulation of I, it is capable of reconstructing a unique synchronous behavior (up to stretch-equivalence).

This means that it can be implemented as a process which is mostly insensitive to (internal and) external propagation delays. This implementation and its context have to agree on activation starts (for instance a physical clock, an external triggering, a procedure call, etc) and on the availability of data, but not necessarily on the presence of signals at the current execution step. This task is controlled by the process that samples continuous signals or select significant values among parameters, read in file or memory, get values in a FIFO or a mailbox, etc.



$$(b|_I)_{\approx} = (c|_I)_{\approx} \quad \Rightarrow \quad b \lesssim c$$

Figure 6: An endochronous design

Definition 16 (endochrony). A process p is endochronous on a (possibly empty) subset $I \subset \text{vars}(p)$ iff

$$\forall b, c \in p,\ (b|_I)_{\approx} = (c|_I)_{\approx} \Rightarrow b \lesseqgtr c$$

We say that a process P is endochronous if it is endochronous on its input signals $\text{in}(P)$. Notice that endochrony implies determinism.

Property 5. An endochronous process is deterministic.

Example 9. The processes `z := x default y` and `z := x when y` are not endochronous. The equation `x := y$1 init v` and all other basic Signal equations (with variables being all distinct) denote endochronous processes.

The processes `Count` (Example 1), `Sampler` (Example 2), `Clock` (Example 3) are not endochronous. Conversely, and despite the fact that its greatest clock is not available from its interface, and that this clock depends upon values, `IsEven` is endochronous.

We can extend a deterministic and non-endochronous process by adding boolean signals and clock constraints. For instance, the following process yields an endochronous extension for `z := x default y`.

```
(| z := x default y
 | cx ^= cy
 | x ^= when cx
 | y ^= when cy |)
```

Endochrony generally requires some clock constraints to be satisfied. Thus, ESTEREL programs or STATECHART are not endochronous. A LUSTRE program is endochronous by design and so are its internal nodes. In this model, a (possibly partial) data-flow function p can be defined as a process satisfying

$$\forall b, c \in p,\ (b|_I)_{\approx} = (c|_I)_{\approx} \Rightarrow (b)_{\approx} = (c_{\approx})$$

where $I \subset \text{vars}(p)$ is the set of input signals. Thus an endochronous process is a data-flow function and the converse is false.

By *exochrony*, we denote the dual of endochrony: input configurations are mastered by the context of the process. An exochronous process is by definition a reactive deterministic process. ESTEREL programs are exochronous.

4.2 Composition properties

Asynchrony We use the partial order relation of relaxation to define the semantics of the asynchronous composition $p \parallel q$ of synchronous processes p and q. Notice that $p \| q \subseteq p \parallel q$.

Definition 17 (asynchrony). Given a behavior b of p s.t. $X = \mathrm{vars}(p)$ and a behavior c of q s.t. $Y = \mathrm{vars}(q)$, the parallel composition of $p \parallel q$ admits the behaviors d that are the relaxations of signals common to b and c (i.e. $I = X \cap Y$).

$$p \parallel q = \left\{ d \,|\, \exists (b,c) \in p \times q, d_{|X\setminus Y} \lesseqgtr b_{|X\setminus Y} \wedge b|_I \sqsubseteq d|_I \wedge d_{|Y\setminus X} \lesseqgtr c_{|Y\setminus X} \wedge c|_I \sqsubseteq d|_I \right\}$$

Flow-invariance The relation of flow-equivalence offers the appropriate criterion for checking the refinement of a system with communication protocols correct. It is, for instance, the property considered in [4] for the refinement-based design of the Ltta protocol in Signal. Flow-invariance consists of ensuring that the refinement of a synchronous specification $p|q$ by an asynchronous implementation $p \parallel q$ preserves the flow of values along signals for any given behavior.

Definition 18 (flow-invariance). The composition of p and q is flow-invariant iff, for all $b \in p|q$, for all $c \in p \parallel q$, $(b|_I)_{\approx} = (c|_I)_{\approx}$ implies $b \approx c$ for I the input signals of $p|q$.

Moreover, flow-invariance is compositional and, as demonstrated in [4], directly amenable to verification using model-checking.

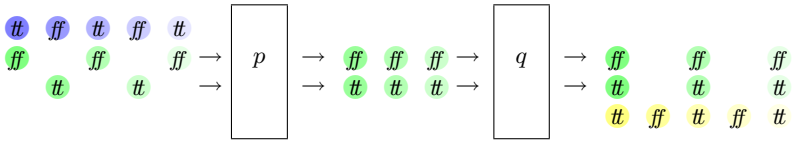


Figure 7: A flow-invariant design

Endo-isochrony The implementation of globally asynchronous and locally synchronous (Gals) architectures in Signal amounts to model a system as a set of *endochronous* components that communicate *endo-isochronously*. Particular protocols such as patient processes or one-place Fifo buffers fall into this category and can be expressed in Signal (see example 10), thanks to polychrony and to the property of stretch and relaxation closure. Endo-isochrony is a much stronger requirement than flow-invariance, derived from the property of endochrony, and is amenable to static verification (as shown section 5.3).

Definition 19 (endo-isochrony). Two processes p and q are endo-isochronous iff p, q and $(p|_I)|(q|_I)$ are endochronous (with $I = \mathrm{vars}(p) \cap \mathrm{vars}(q)$).

The properties of endochrony and endo-isochrony allow for the compositional design of distributed designs starting from polychronous specifications.

Property 6 (endo-isochrony). If p and q are endo-isochronous then the composition of p and q is flow-invariant.

Example 10. The following process implements an alternating input/output one-place buffer protocol (it is parameterized over the type t of the data and over the initial value i stored in the buffer's register).

```
process Buffer = {type t; t i}(? t x ! t y)
   (| y := Current {i}(x)
    |      Alternate (x, y)
    |) where process Current = {t i} (? t x ! t y)
               (| y := (x cell ^y init i) when ^y
                |)
             process Alternate = (? t x, y ! )
               (| x ^= when b
                | y ^= when not b
                | b := not (b$1 init false)
                |) where boolean b;
             end;
end;
```

tags	·	·	·	·	·	·	·	·	·	·
x	1		2		3		4		5	
y		1		2		3		4		5

It is implemented in two steps. The sub-process Current (modeling memory requirements) stores any value incoming from x and loads it into y upon demand (i.e. at the clock of y). The sub-process Alternate (modeling protocol constraints) alternates the clocks of x and y by synchronizing them to the different values of a boolean flip-flop signal b. Notice that the input x and the output y of the process Buffer are isomorphic.

4.3 Related works

The first formal address of *desynchronization* can be found in [10], where precise relations between well-clocked synchronous functional programs and the subset of Kahn-networks are established, and shown to be amenable to buffer-less evaluation. In [11], the author considers the distribution of synchronous automata on asynchronous networks using FIFO-buffered broadcast communications. In [8], a model for the distribution of synchronous programs on distributed architectures is introduced which uses low-level non-blocking one-place buffers. In [2], the transformation of synchronous designs on distributed architecture is further developed in the context of SIGNAL. In [3], an analysis of the links between synchrony and asynchrony is presented in the context of synchronous transition systems (STS) and the compositional property of isochrony introduced (a pair of processes is isochronous iff its synchronous and asynchronous compositions admit the same traces). Notice that flow-invariance implies isochrony.

The semantics of asynchrony of definition 17 is of comparable expressive power to that proposed in [3] for synchronous transition systems. It is expressed within the

tagged model of computation of [19] and, unlike [3], equipped with partial order and equivalence relations that provide a structure of semi-lattice. It encompasses finite FIFO buffer implementations, as considered in [10,11], as well as low-level non-blocking one-place buffers implementations, a considered in [8].

In [9], a theory of latency-insensitive protocols (LIP) is presented as a foundation of a new methodology to design very large digital systems by assembling blocks of existing *intellectual property* (IP). It is appealing to cast the tagged model of polychronous signals in the theory of LIP to demonstrates how correct-by-construction polychronous designs can be implemented in this framework.

In the theory of LIP, a process p may accept *stall moves* (a stall move consists of stretching a signal s of a given behavior b by one logical instant) and respond by the appropriate *procrastination* effect (the other signals s' of the behavior b will respond to the delay of s by making coordinated stall moves).

The mechanism of causally related stall moves is made explicit (and generalized to polychronous signals) in the relation of stretching $\sqsubseteq$ (definition 4). A stretch corresponds to a slide of tags in a given behavior: if an event e is stalled, then all other events e', causally related to e by the partial order relation $\leq$) will slide. As a result, SIGNAL processes are *patient*, in the sense of [9].

Definition 20 (patience [9]). A patient process p is a process such that, for all behaviors $b \in p$, for all and for all signals $x \in \text{vars}(b)$ s.t. $b(x) = (t_j, v_j)_{j>0}$, if a signal $s' = (t'_k, v_k)_{k>0}$ differs from s by one stall move (i.e. there exists $l > 0$ s.t. $t_j = t'_k$ for all $j = k < l$ and $t_l \leq t'_l$) then there exists $b' \in p$ such that $b'(x) = s'$.

The (multi-clocked) notion of flow-equivalence relates to the (single-clocked) notion of latency-equivalence of Carloni et al. [9]. Two signals are latency-equivalent iff they present the same values in the same order.

Definition 21 (latency-equivalence [9]). Let $(t_i)_{i>0}$ and $(t'_i)_{i>0}$ be two chains of $\mathcal{T}$ s.t. for all $j, k > 0$, $j \leq k$ iff $t_j \leq t_k$ and $t'_j \leq t'_k$. The signals $s = (t_i, v_i)_{i>0}$ and $s' = (t'_i, v'_i)_{i>0}$ are latency equivalent iff for all $i > 0$, $v_i = v'_i$.

Notice that, if $s \approx s'$ then s and s' are latency equivalent. Flow-equivalence extends the property of latency-equivalence to multi-clocked systems: the theory of LIP considers synchronous processes equipped with totally ordered time tags (i.e. single-clocked systems). This hypothesis corresponds to the low-level behavior of circuits (where silence, i.e. τ events, corresponds to stall moves or "don't care" moves). The tagged model of polychronous signals considers a more general structure of partially ordered semi-lattice.

In SIGNAL, the implementation communication media proposed in [9]: buffers, equalizers and relay processes; allows for the correct-by-construction design of latency-insensitive processes starting from endochronous (patient) IP blocks.

5 System design with polychrony

The SIGNAL model provides a design methodology which encompasses compilation and distribution. The implementation of SIGNAL supports this broad spectrum of utilization by providing accurate data-structures and algorithms to manipulate and transform the model of an application, from the specification of early requirements and property to the generation of sequential and distributed code.

The media for reasoning on a system under design in SIGNAL consists of a synchronization relations, represented as clock hierarchies (section 5.1), and causality relations, represented by conditional dependency graphs (section 5.2).

These tools and the analysis and transformation techniques exposed in present section yield a seamless methodology that capture system design from its early specification stages (requirements can be expressed by synchronization and causality properties in SIGNAL) down to late deployment stages (hierarchized and serialized designs can be expressed in SIGNAL).

5.1 Synchronization relations

In SIGNAL, the presence of a value along a signal x at a given tag is denoted by its clock $\hat{x}$. Referring to the tagged model of polychronous signals (section 3.1), the clock of a signal represents the set of tags at which the signal holds a value. This object can easily be represented in the syntax of SIGNAL. For a given signal x, we write $\mathsf{event}\, x$ or $\hat{x}$ for the signal that holds the value true iff x is present: its clock.

$$\begin{array}{rl} x : & (t_1, v_1)\,(t_2, v_2)\ (t_3, v_3)\ldots \\ \mathsf{event}\, x : & (t_1, \mathit{tt})\ \ (t_2, \mathit{tt})\ \ (t_3, \mathit{tt})\ \ \ldots \end{array}$$

Notice that both a system specification and its clock abstraction are objects which can both be represented and manipulated in SIGNAL in order to produce and represent successive design transformation, from early requirement specifications to late sequential and distributed code generation. For instance, the process event can be defined in SIGNAL as follows.

```
process event = (? x ! y ) (| y = (x=x) |)
```

A synchronization relation is described by an equation between signal clocks. Given two signals x and y, we write $\mathsf{synchro}\, x\, y$ (or $\hat{x} = \hat{y}$) to synchronize them. The process synchro can itself be defined in SIGNAL as follows.

```
process synchro = (? x, y !) (| (event x)=(event y) |)
```

In order to enable reasoning on control in a system, we write $[x]$ for the clock at which a boolean signal x (resp. $[\neg x]$ holds the value true (resp. false). We naturally have the following relations between boolean signal clocks. These relations can themselves be represented in SIGNAL. We write 0 for the empty clock. The grammar

of clock expressions h and clock relations H is defined by induction as follows.

$$\begin{array}{rcll} h & ::= & 0 \mid \hat{x} \mid [x] \mid [\neg x] \mid h \hat{\times} h' \mid h \hat{+} h' \mid h \hat{-} h' & \text{(clocks)} \\ H & ::= & \emptyset \mid h = h' \mid h < h' \mid D \cup H' \mid \exists x.H & \text{(equations)} \end{array}$$

The domain of clocks $(\mathcal{H}, \hat{+}, \hat{\times}, \hat{-}, 0)$ forms a semi-lattice of union operator $\hat{+}$, intersection $\hat{\times}$, difference $\hat{-}$ and neutral element 0. In particular, we have the following remarkable properties between clocks and sampling that $\hat{x} = [x] \hat{+} [\neg x]$ and that $[x] \hat{\times} [\neg x] = 0$ for all signal names x.

In the SIGNAL compiler, clock relations are reconstructed from a given specification by making use of the following clock inference procedure. Note that existential quantifiers can be eliminated by replacing equations, e.g. $\exists x.\{\hat{y} = \hat{x} \hat{+} \hat{z}\}$, by inequations, e.g. $\{\hat{y} \geq \hat{z}\}$.

$$\begin{array}{rcl} x = \mathsf{pre}\, v\, y : \{\hat{x} = \hat{y}\} & & \\ x = y \,\mathsf{when}\, z : \{\hat{x} = \hat{y} \hat{\times} [z]\} & & \\ x = y \,\mathsf{default}\, z : \{\hat{x} = \hat{y} \hat{+} \hat{z}\} & & \\ P \mid Q : H \cup H' & \text{iff } P : H \text{ and } Q : H' & \\ P/x : \exists x.H & \text{iff } P : H & \end{array}$$

In order to establish the property that assesses the correctness of the inference system $P : H$, we define an interpretation of the set of clock relations H by the set of behaviors that satisfy these relations. We write this interpretation $[\![H]\!]$. A clock expression h is interpreted as the set of tags that characterize it for a given behavior b, written $b|_h$.

$$\forall b \in \mathcal{B}, \quad b|_{\hat{x}} = \begin{cases} \text{tags}(b(x)), & x \in \text{vars}(b) \\ \emptyset, & x \notin \text{vars}(b) \end{cases} \qquad b|_{[x]} = \begin{cases} \{t \in \text{tags}(b(x)) \mid b(x)(t) = tt\}, & x \in \text{vars}(b) \\ \emptyset, & x \notin \text{vars}(b) \end{cases}$$

$$b|_{[\neg x]} = b|_{\hat{x}} \setminus b|_{[x]} \qquad b|_{h \hat{\times} h'} = b|_h \cap b|_{h'} \qquad b|_{h \hat{+} h'} = b|_h \cup b|_{h'} \qquad b|_{h \hat{\setminus} h'} = b|_h \setminus b|_{h'}$$

Then, a set of clock relations H is interpreted by the set $[\![H]\!]$ that contains all the behaviors which satisfy it.

$$\begin{array}{c} [\![h = h']\!] = \{b \in \mathcal{B} \mid b|_h = b|_{h'}\} \\ [\![h < h']\!] = \{b \in \mathcal{B} \mid b|_h \subset b|_{h'}\} \\ [\![H \cup H']\!] = [\![H]\!] \cup [\![H']\!] \\ [\![\exists x.H]\!] = [\![H]\!]_{/x} \end{array}$$

We show that the behaviors $[\![P]\!]$ of a process P are contained in the interpretation $[\![H]\!]$ of its clock constraints H.

Property 7. If $P : H$ then $[\![P]\!] \subseteq [\![H]\!]$

Hierarchization The SIGNAL compiler uses the data structure of clock relations H to built a hierarchy of clocks. This hierarchy, which can be informally seen as

a forest of trees, renders the partial order relations between the clock of all signals defined in a given specification. Two signals x and y appear at the same node of the hierarchy iff they are synchronous, i.e. $\hat{x} = \hat{y}$. A signal x is placed in a branch under a signal y iff it is a down-sampling of y, i.e. $\hat{x} < \hat{y}$.

The construction of the hierarchy of clocks, introduced by Amabegnon et al. in [1], constitutes the core of the SIGNAL compiler. It consists of first building elementary trees and then of merging the root of those trees according to clock relations specified in the remainder of the constraint set H. The hierarchization algorithm proposed by Amabegnon et al. in [1] possesses a couple of remarkable properties:

- It produces a canonical representation of the clock constraint set C.
- It determines whether a design is endochronous (see definition 16): if the hierarchy of a process forms a tree then the order of evaluation of the equations of P is recursively determined by the clocks from top to the bottom of the tree.

A sufficient condition for the existence of a canonical representation of H for a given design p is the existence of a master clock x for p, i.e., a signal x such that H implies $\hat{x} > \hat{y}$ for all other signals y of p. Indeed, in a (deterministic) SIGNAL design, output signal clocks are defined modulo the input signal clocks (for pre, $\hat{x} = \hat{y}$; for pre, $\hat{x} = \hat{y} \hat{+} \hat{z}$; for pre, $\hat{x} = \hat{y} \hat{\times} [z]$). Therefore, the existence of a master x guarantees the computability of all output clocks starting from the input clocks.

Example 11. In order to give an informal assessment of this result (formalized section 5.3), let us reconsider the process Sampler (example 1) and determine its synchronization relation using the inference system $P : H$. The SIGNAL compiler produces the following synchronization constraints for the process Sampler.

Its interpretation is that the clock of val is the union of the reset and tick signal clocks (expression `CLK_val := reset ^+ tick`). The inferred clock CLK_1 (defined by the expression `CLK_1 := reset ^+ alarm`) is that at which the counter val is reset to 0. The inferred clock CLK_2 (defined by the expression `CLK_2 := CLK_val ^- CLK_1`) is that at which the counter val is incremented. The process is endochronous since we have that $\hat{\text{val}} = \hat{\text{mod}}$, $\hat{\text{val}} \geq \hat{\text{tick}}$, $\hat{\text{val}} \geq \hat{\text{reset}}$ and $\hat{\text{val}} \geq \hat{\text{alarm}}$.

```
(| CLK_val := reset ^+ tick
 | CLK_1   := reset ^+ alarm
 | CLK_2   := CLK_val ^- CLK_1
 |)
```

5.2 Causality relations

A complementary feature provided by SIGNAL allows to express control flow. The conditional data-flow graph in the SIGNAL compiler handles this control flow as causal relations between signals at given clocks: the relation $x \rightarrow^h y$ specifies that

x is a cause of y at the clock of h. Causal relations are transitive and distributive w.r.t. clocks.

$$G ::= \emptyset \,|\, x \rightarrow^h y \,|\, G \cup G' \,|\, G \cap G' \,|\, G \setminus G' \,|\, \exists x.G$$

The conditional data-flow graph of a process is subject to the following distribution rules, which allow to construct its closure $\overline{G}$.

$$\begin{array}{rcl} \forall x, y \, \forall h, h' \;\; x \rightarrow^h y \cup y \rightarrow^{h'} z & \Rightarrow & x \rightarrow^{h \hat{\times} h'} z \\ x \rightarrow^h y \cup x \rightarrow^{h'} y & \Rightarrow & x \rightarrow^{h \hat{+} h'} y \end{array}$$

In the SIGNAL compiler, the data-structure of conditional data-flow graph G is used to schedule and serialize the equations of a design into elementary operations so as to produce optimized sequential code (for an acyclic graph) or distributed code (for a clustering of this graph into acyclic sub-graphs).

The presence of a cycle in the data-flow graph renders a causality loop of fixed-point equation that cannot be implemented by a single operation (e.g. $x = x + 1$ defines the immediate value of x by the solution of a fixed-point equation).

The advantage of conditioning causal relations by clocks is that spurious cycles can easily be detected: $x \rightarrow^h x$ is spurious iff H implies $h = 0$ (if, e.g., $c = [x] \hat{\times} [\neg x]$). In the SIGNAL compiler, causal relations are reconstructed from a given specification by making use of the following inference procedure (existential quantifiers can be eliminated by erasing local signals, e.g. $\exists x.(y \rightarrow^h x \cup x \rightarrow^{h'} z)$, from the transitive closure, e.g. $y \leftarrow^{h \hat{\times} h'} z$, of a graph).

$$\begin{array}{rcl} x = \mathsf{pre}\, v\, y : \emptyset & & \\ x = y \,\mathsf{when}\, z : y \rightarrow^{[z]} x & & \\ x = y \,\mathsf{default}\, z : y \rightarrow^{\hat{y}} x \cup z \rightarrow^{\hat{z} \hat{\setminus} \hat{y}} x & & \\ P \,|\, Q : G \cup G' & & \text{iff } P : G \text{ and } Q : G' \\ P/x : \exists x.G & & \text{iff } P : G \end{array}$$

Interpretation of causality graphs The interpretation of causality graphs consists of a refinement of the structure $(\mathcal{T}, \leq)$ of polychronous signals to render the relative schedule of simultaneous events within an instant (i.e. events which carry the same tag). This refinement is obtained by considering a scheduling pre-order $(\mathcal{G}, \rightarrow)$. The domain of scheduling tags $\mathcal{G}$ is isomorphic to $\mathcal{X} \times \mathcal{T}$. We assume a bijection σ of $\mathcal{X} \times \mathcal{T} \rightarrow \mathcal{G}$ and write $t_x = \sigma(t, x)$ for the relative schedule of a signal named x at the instant t.

$$\sigma : \mathcal{X} \times \mathcal{T} \rightarrow \mathcal{G} \qquad (x, t) \mapsto t_x$$

It is subject to a pre-order relation $\rightarrow$ that denotes scheduling: $t_x \rightarrow t'_{x'}$ means that $t'_{x'}$ cannot happen before t_x. The scheduling relation is chosen so as to satisfy a containment relation w.r.t. $(\mathcal{T}, \leq)$.

$$\forall x \in \mathcal{X}, \, \forall s \in \mathcal{S}, \, \forall t, t' \in \mathrm{tags}(s), \, t < t' \Rightarrow t_x \rightarrow t'_x$$

The definition of the domains of behaviors $\mathcal{B}$ and processes $\mathcal{P}$, of the relations of stretching $\leq$ and relaxation $\sqsubseteq$ carry over this refinement.

Definition 22 (stretching). A behavior c is a stretching of b, written $b \leq c$, iff $\text{vars}(b) = \text{vars}(c)$ and there exists a bijection $f \in \mathcal{T} \to \mathcal{T} + \mathcal{G} \to \mathcal{G}$ that is

monotonic along all chains $\forall C \in \mathcal{C}$, $\forall t \in C$, $t \leq f(t)$

strictly increasing:

$\forall t, t' \in \mathcal{T}$, $t < t' \Leftrightarrow f(t) < f(t') \wedge \forall x \in \text{vars}(b)$, $t_x \to t'_x \Leftrightarrow f(t_x) \to f(t'_x)$

satisfying, for all $x \in \text{vars}(b)$,

$\forall t \in \text{tags}(b(x)), f(t)_x = f(t_x)$

$b(x)(t) = c(x)(f(t))$

$\text{tags}(c(x)) = f(\text{tags}(b(x)))$

The denotation of restriction P/x, synchronous composition $P \,|\, Q$ and parallel composition $P \parallel Q$ extends to the refined domain of signals starting from the following definitions for primitive equations.

$$[\![x = \mathsf{pre}\, v\, y]\!] = \left\{ b \in \mathcal{B}|_{x,y} \;\middle|\; \begin{array}{l} \text{tags}(b(x)) = \text{tags}(b(y)) = C \in \mathcal{C}\backslash\emptyset,\ b(x)(\min(C)) = v \\ \forall t \in C \setminus \min(C),\ b(x)(t) = b(y)(\text{pred}_C(t)) \end{array} \right\} \cup \{\mathsf{0}|_{x,y}\}$$

$$[\![x = y\ \mathsf{when}\ z]\!] = \left\{ b \in \mathcal{B}|_{x,y,z} \;\middle|\; \begin{array}{l} \text{tags}(b(x)) = \{t \in \text{tags}(b(y)) \cap \text{tags}(b(z)) \,|\, b(z)(t) = tt\} \\ \forall t \in \text{tags}(b(x)),\ b(x)(t) = b(y)(t) \wedge t_y \to t_x \wedge t_z \to t_x \end{array} \right\}$$

$$[\![x = y\ \mathsf{default}\ z]\!] = \left\{ b \in \mathcal{B}|_{x,y,z} \;\middle|\; \begin{array}{l} \text{tags}(b(y)) \cup \text{tags}(b(z)) = \text{tags}(b(x)) = C \in \mathcal{C} \\ \forall t \in \text{tags}(b(y)),\ b(x)(t) = b(y)(t) \wedge t_y \to t_x \\ \forall t \in C\backslash\text{tags}(b(y)),\ b(x)(t) = b(z)(t) \wedge t_z \to t_x \end{array} \right\}$$

to which we can add explicit scheduling specifications:

$$[\![x \to^h y]\!] = \{b \in \mathcal{B}|_{x,y,\text{vars}(h)} \,|\, \forall t \in b|_h,\ t_x \to t_y\}$$

$$[\![x = \mathsf{pre}\, tt\, y]\!] \ni \left(\begin{array}{llllllll} y & \mapsto & ff & tt & ff & tt & & \cdots \\ x & \mapsto & tt & ff & tt & ff & tt & \cdots \end{array} \right)$$

$$[\![x = y\ \mathsf{default}\ z]\!] \ni \left(\begin{array}{llllllll} z & \mapsto & ff & & ff & tt & & \cdots \\ y & \mapsto & tt & ff & \downarrow & ff & tt & \cdots \\ & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \cdots \\ x & \mapsto & ff & ff & ff & tt & tt & \cdots \end{array} \right)$$

$$[\![x = y\ \mathsf{when}\ z]\!] \ni \left(\begin{array}{llllllll} z & \mapsto & tt & tt & & & tt & \cdots \\ y & \mapsto & ff & & ff & tt & tt & \cdots \\ & & & & & & \downarrow & \cdots \\ x & \mapsto & & & & & tt & \cdots \end{array} \right)$$

$$[\![y \to^{\hat{z}} x]\!] \ni \left(\begin{array}{llllllll} z & \mapsto & ff & & & tt & & \cdots \\ y & \mapsto & tt & ff & & & tt & \cdots \\ & & \downarrow & & & & & \cdots \\ x & \mapsto & ff & & ff & & tt & \cdots \end{array} \right)$$

Figure 8: Causality specifications for delay, sampling and merge

Property 8 relates the interpretation of scheduling pre-orders in the denotational semantics $[\![P]\!]$ of processes and its abstraction using causality graphs.

Property 8. If $P:G$ then $[\![P]\!] \subseteq [\![G]\!]$

5.3 Static verification of global invariants

In the SIGNAL compiler, causality graphs are used to determine a specification of the scheduling between different equations at a given clock. In particular, it allows to check whether a process P (section 3.4) is deterministic. We write $H \Rightarrow H'$ (resp. $G \Rightarrow G'$) iff H' is deductible from H (resp. G' from G) and write $\exists x.H \Rightarrow H'$ if $H[y/x] \Rightarrow H'$ for $y \notin \text{vars}(H)$.

Property 9. A process P such that $P:H$ and $P:G$ is deterministic if G is acyclic (i.e. for all x, if $G \Rightarrow x \rightarrow^h x$ then $H \Rightarrow h = 0$) and H is guarded (i.e. for all x, if $H \Rightarrow \hat{x} = h$ then $H \Rightarrow h = h'$ with $\text{vars}(h') \subseteq \text{in}(P)$).

The property 10 checks that a deterministic process P is endochronous by ensuring that the clock of every signal y of P is computable starting from the master clock of the process P (that of the input signal x which is the most frequent, i.e. $\hat{y} \leq \hat{x}$). This allows for a unique flow of control to be iteratively reconstructed from the value of the boolean signals present at a given tag.

Property 10. A deterministic process P is endochronous if $P:H$ and there exists $x \in \text{in}(P)$ s.t., for all $y \in \text{vars}(p)$, $H \Rightarrow \hat{y} \leq \hat{x}$.

The property 10 can be used to determine whether a given process is a compilation unit, and allow for its separate compilation as a given IP of a system. The property 10 can also be used to determine whether two endochronous processes P and Q are endo-isochronous, and allow for the compositional integration of separately compiled IPs within a distributed architecture.

5.4 Verification of system designs

The equational style of SIGNAL and its relational model yields to considering polynomial dynamical equations over $\mathbb{Z}/3\mathbb{Z}$ to characterize and manipulate process behaviors. Nonetheless, related symbolic model-checkers can equivalently be used to verify SIGNAL properties. In the model-checker SIGALI [20], the set of solutions (states and events) of a process is modeled by a system of polynomial equations.

In the $\mathbb{Z}/3\mathbb{Z}$ algebra, the status of a signal x is rendered by three possible values: 0 iff absent, 1 iff present with the value true and -1 iff present with the value false. The clock of a signal x is the square x^2 of x: 1 iff present, 0 iff absent. Non-boolean

signals are abstracted by clock equations that represent their synchronization relations (e.g. two synchronous integer signals x and y are represented by the equation $x^2 = y^2$). Each equation of a process P is encoded by a polynomial equation. For instance, The encoding of delay makes use of a state variable, noted σ, to model a state transition. The first equation describes the next value σ' of σ; it is equal to y iff y is present and to σ otherwise (i.e. $\sigma.(1-y^2)$). The second equation defines the value of x by that of σ when y is present. The last equation defines the initial value of the state variable σ by the given initialization constant v.

$$\begin{array}{rcl} [\![x = \mathsf{not}\, y]\!] & = & y = -x \\ [\![x = y\ \mathsf{when}\ z]\!] & = & x = y.(-y - y^2) \\ [\![x = y\ \mathsf{default}\ z]\!] & = & x = y + (1 - x^2).y \end{array} \qquad [\![x = \mathsf{pre}\, v\, y]\!] = \left\{ \begin{array}{rcl} \sigma' & = & y + \sigma.(1 - y^2) \\ x & = & \sigma.y^2 \\ \sigma_0 & = & v \end{array} \right.$$

The composition of elementary $\mathbb{Z}/3\mathbb{Z}$ equations forms the polynomial dynamical system of a process. This system consists of three remarkable components, that manipulate a vector $X \in (\mathbb{Z}/3\mathbb{Z})^n$ of n state variables and a vector $Y \in (\mathbb{Z}/3\mathbb{Z})^m$ of m event variables.

- $X' = P(X, Y)$ is the evolution equation. It is a vectorial function from $(\mathbb{Z}/3\mathbb{Z})^{n+m}$ to $(\mathbb{Z}/3\mathbb{Z})^n$ which characterizes the dynamical aspects of the system: the evolution of state variables in time.
- $Q(X, Y) = 0$ is the constraint equation. It consists of l equations $Q_1, \ldots Q_l$ which characterize the invariants of the system (usually clock equations).
- $Q_0(X) = 0$ is the initialization equation. It consists of n equations which characterize the initial values of the state variables $X_1, \ldots X_n$.

A polynomial dynamical system can equivalently be interpreted as a finite transition system or as a Kripke structure. The initial states of the corresponding automaton are the solutions of the initialization equation Q_0. When the system is in a given state $x \in (\mathbb{Z}/3\mathbb{Z})^n$, any event $y \in (\mathbb{Z}/3\mathbb{Z})^m$ satisfying the constraint equation $Q(x, y)$ can fire a transition of the automaton to a state x' characterized by the evolution equation $x' = P(x, y)$. To compute the properties of polynomial dynamical systems, SIGALI makes use of relations on varieties and ideals in the the quotient ring of polynomial functions $A[X, Y] = \mathbb{Z}/3ZZ[X, Y]/(X^3 - X, Y^3 - Y)$. using morphisms and co-morphisms ($A[X, Y]$ characterizes the set of polynomials in which the degree of each variable does not exceed 2: $X^3 = X$)We briefly state the most common ones: liveness and invariance. Other properties, such as reachability and attractivity can be defined starting from the notions of liveness and invariance.

Definition 23 (liveness and invariance). A state x is *alive* iff there exists an event y s.t. $Q(x, y) = 0$. A set of states V is alive iff every state of V is alive. A system is alive iff for all (x, y) s.t. $Q(x, y) = 0$, $P(x, y)$ is alive. A set of states E is *invariant* iff for every state $x \in E$ and every event y admissible in that state (i.e. s.t. $Q(x, y) = 0$) such that the state $P(x, y)$ is in E.

6 Design methodologies

We now have at our disposal the model of polychronous signals equipped with decision procedures for checking formal design properties in order to explore its practical use in the context of bottom-up and top-down design methodologies.

6.1 Refinement-based design methodology

To every step of a refinement-based engineering methodology corresponds an initial specification P (a process) and a revised one, Q, that results of defining new intermediate variables by adding equations to P. This refinement may either result in latency (stretching or relaxation) w.r.t. the initial runtime behavior of P, or incur an oversampling of P (by refining its time scale). Thanks to the analysis of synchronization and causal relations implemented in SIGNAL, one can associate the initial specification P (resp. its upgrade $P \,|\, Q$) to a hierarchy of clocks H and a data-flow graph G (resp. H' and G').

Statically checking semantics-preserving refinement Checking that the upgrade of P by $P \,|\, Q$ is a refinement (i.e. a non-regressive upgrade) amounts, first, to proving global safety requirements of the upgrade (endochrony, cycle-freedom) and, second, to showing that the graph G' contains the graph G (i.e. $G' = G \cup G'$ for some G') and that the hierarchy H' implies the hierarchy H (i.e. $H' \Rightarrow H$). Since H and H' are boolean equations, the problem of checking the property $H' \Rightarrow H$ can be expressed by encoding H and H' by series of polynomials $(P_i(X) = 0)_{i \in I}$ and as $(Q_j(X) = 0)_{j \in J}$ (by considering the vector X formed by the clocks H and H'). Let $\oplus$ be the boolean operator $a \oplus b = a + b + a.b, \forall ab$. A property of $\oplus$ is that $P_1(\boldsymbol{x}) = 0$ and $P_2(\boldsymbol{x}) = 0$ iff $(P_1 \oplus P_2)(\boldsymbol{x}) = 0$. Using $\oplus$, the system of boolean equations reduces to $P(\boldsymbol{x}) = 0$ and $Q(\boldsymbol{x}) = 0$ where $P = \oplus_{i \in I} P_i$ et $Q = \oplus_{j \in J} P_j$. Let V and W be the solutions of P and Q, we have that:

$$V \subseteq W \Leftrightarrow \forall X.(1 - P(X)).Q(X) = 0$$

This demonstrates that refinement-checking is amenable to the resolution of a BDD which corresponds to the boolean function $F(X) = (1 - P(X)).Q(X)$. This technique is implemented in the SIGNAL compiler.

Model-checking semantics-preserving refinement Although static checking may be a convenient tool to prove the refinement of a design correct w.r.t. an initial specification correct in most cases, the relation of flow-equivalence offers a more precise metrics. It is, for instance, the property considered in [4] for the refinement-based design of the LTTA protocol in SIGNAL. The invariance of a design refinement w.r.t. the relation of flow equivalence is directly amenable to verification using model-checking (in the case of reactive and polychronous communication protocols such as, e.g. finite FIFO buffers).

Transformation-based design refinements To illustrate transformation-based refinement, let us go through the following process (avse), which returns $y = a + b$ when both a and b are present, $y = 2 * b$ when only b is present, and its previous value otherwise.

```
process avse = (? integer a, b ! integer y )
   (|  x := a default b
    |  y := ((x when ^b) + b ) cell ^x
    |) where integer x;
end;
```

To obtain a flow-preserving distributed implementation of the process avse, we start by building a clocked refinement Cavse of avse.

```
process Cavse = (? integer a, b ! integer y )
   (|  x   := a default b
    |  C_a := (true when ^a) default false
    |  C_b := (true when ^b) default false
    |  C_x := C_a or C_b
    |  y   := ((x when C_b) + b) cell ^x
    |) where integer x; boolean C_a, C_b, C_x;
end;
```

Cavse has the same behavior as avse, since C_a, C_b and C_x (the boolean clocks of a, b, x and y) are internal signals. Let S be the resulting sampler (definition 9, the construction of sampler is available in the POLYCHRONY environment). The original equations are (syntactically) split in two sub-processes, Default and addb, whose composition is endo-isochronous. The resulting process, DCavse, equals the behaviors of Cavse and of avse.

```
process DCavse = (? integer a, b ! integer y )
   (|  (x, C_b, C_x) :=  Default (a,b);
    |             y :=  addb(b, x, C_b, C_x)
    |) where integer x; boolean C_b, C_x;
            process Default = (? integer a, b ! integer x; boolean C_b, C_x )
              (| x   := a default b
               | C_a := (true when ^a) default false
               | C_b := (true when ^b) default false
               | C_x := C_a and C_b
               |) where boolean C_a;
            end;
            process  addb = (? integer b, x; C_b, C_x ! integer y )
              (| y  := ((x when C_b) + b ) cell C_x
               | b  ^= when C_b
               | x  ^= when C_x
               | C_b ^= C_x
               |)
end;
```

Notice that addb is endochronous. In the current implementation of syntactic distribution in POLYCHRONY, Default is also turned into an endochronous process. Because the composition of Default and addb is endochronous we can model communication time consumption by equally buffering b, x, C_b, C_x, provided that the introduced buffering mechanism preserves flows (it could for instance be the process Buffer of example 10). To facilitate the specification of such protocols, SIGNAL offers a sliding window operator, that allows for the specification bounded FIFO protocols. Stepping further toward a distributed implementation leads us to consider different buffering mechanisms and equip addb with a synchronous/asynchronous interface that rebuilds input configurations. Such an interface makes addb reactive (in hardware this role is played by holding values and correct clock management). It preserves flow equivalence, provided that buffers are long enough. An example of this transformation is given below.

```
process AsyncAdd = (? integer a, b ! integer y )
  (| m_a       := a cell ^b
   | m_b       := b cell ^a
   | react     := true when (^a default ^b)
   | sync      := true when (^a when ^b)
   | argWaits := ( (false when C_y) default react ) $1 init false
   | C_y       := sync default (argWaits when react)
   | y         := Adder(m_a when C_y, m_b when C_y)
   |) where integer m_a, m_b;
            boolean argWaits, sync, C_y;
            process Adder = (? integer a, b ! integer y ) (| y := a + b |)
end;
```

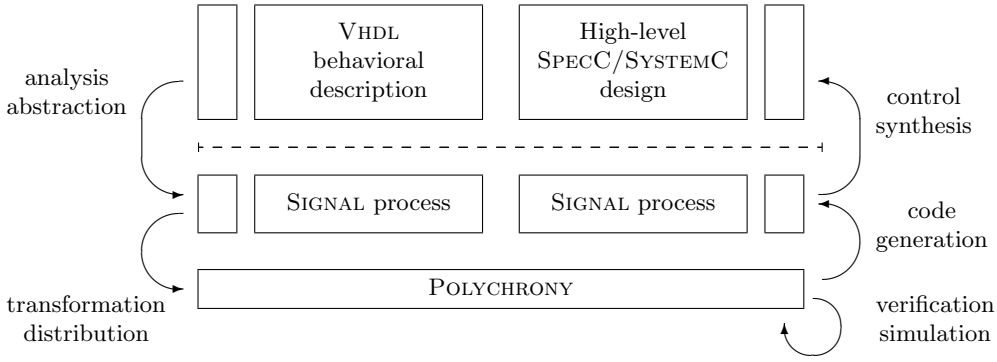
AsyncAdd is a reactive refinement for the synchronous Adder that accepts bounded desynchronizations of a and b (i.e. no more than one occurrence of a between two occurrences of b and conversely). Such a requirement may be given by adding the specification `(| notTwice(^a,^b) | (notTwice(^b, ^a)) |)` to the interface of AsyncAdd, where the process `notTwice(x, h)` is defined by

```
process notTwice = (? event x,h )
   (| oneX := (false when h default true when x) $1 init false
    | x    ^= x when not oneX
    |) where boolean oneX;
end;
```

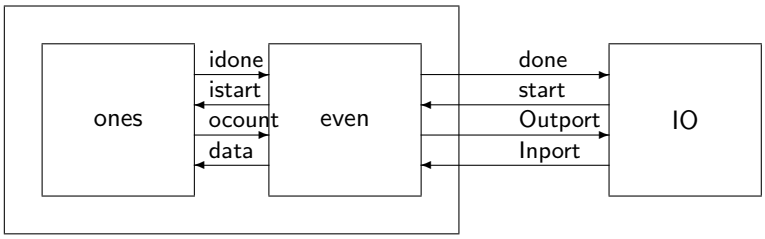
The correctness of the top-down refinements of the process avse is guaranteed by the relation of flow-equivalence. Let us discuss briefly some of its other properties. Notice that the signal y in AsyncAdd is defined by `y := Adder(`m_a `when` C_y`,` m_b `when` C_y`)`. This means that Adder is invoked only if needed. This can be a drawback in hardware design: this incurs adding control to combinatorial parts. If AsyncAdd is known as being stateless, the semantics of SIGNAL allows to substitutes it by the equivalent `y := Adder(`m_a, m_b`) when` C_y. Conversely, in DCavse, communication between Default and addb can be minimized by sending x only when it is needed i.e. at the clock b.

6.2 Component-based design methodology

The installed base of intellectual property (IP) calls for proposing appropriate mathematical models and methodological approaches to integrate systems by adapting existing IPs with new services within complex integrated architectures, in the face of rising technological complexity, increasing performance requirements, shortening time to market demands for electronic equipments. The main drivers for high-level design languages such as SpecC or SystemC lies in the object-oriented structuring of system components and in the ease of programming of a C-like syntax and (informal) semantics. The main benefit of considering a model of communicating synchronous processes for high-level design in C-like programming languages is to provide a semantical backbone for design, on which verification and optimization techniques can then be plugged in.



The abstraction of an architecture consisting of existing IP blocks in Signal consists of representing it as a desynchronized network of synchronous processes using the tagged model of polychronous signals. Let us informally sketch the feasibility of this process by considering a small SpecC example: an even-parity checker. We only detail the most characteristic thread of this architecture.



The behavior ones determines the parity of an input data received along the Inport. Upon receipt of the start notification, it repeatedly shifts the data right until 0 is reached. The resulting ocount (output count) is sent along the Outport together with the notification done.

The translation of the behavior ones in SIGNAL consists, first, of decomposing the syntactic structure of the SPECC program into an intermediate representation that renders the imperative structure of the original program together with its most characteristic features (threads, locks, interrupts, etc).

In this structure, each thread consists of a sequence of blocks (critical sections) delimited by wait and notify synchronization statements. Within such blocks, basic control structures are then encoded.

A method call or a basic operation, e.g. $x = y+1$, is encoded by an equation, e.g. either $x = y\$1 + 1$ when c (when y references a value computed during the previous transition in this block) or $x = y + 1$ when c (if it has already been computed in the same transition), conditioned by an activation clock c. A conditional statement, e.g. if x then P else Q, is encoded by constraining the clock of P by x and that of Q by not x. Internal while loops are encoded by over-sampling (see example 4).

Interrupts are rendered by boolean signals that tell whether or not they are raised during a computation. An interrupt conditions the activation clock of subsequent equations in the control flow graph; if it escapes the scope of the method in which it is raised, it becomes an output signal of the process that encodes the method in order to propagate in the context of use of that method.

The encoding of the behavior ones in SIGNAL yields a process in which the clock of input/output signals are synchronized to input/output notification events. The process ones consists of one critical section. The internal while loop is encoded by an over-sampling sub-process.

```
behavior ones(in unsigned int Inport, out unsigned int Outport,
              in event start, out event done)
{
        void main(void) {
                unsigned int data, ocount, mask, temp;
                while (1) { wait(start);
                            data = Inport;
                            ocount = 0;
                            mask = 1;
                            while (data != 0) { temp = data & mask;
                                                ocount = ocount + temp;
                                                data = data >> 1;
                            }
                            Outport = ocount;
                            notify(done);
}}};
```

Exception mechanisms are rendered by boolean signals that determine whether or not an interrupt is raised a computation. Exception signals condition the activation clock of subsequent equations in the control flow graph. If an interrupt escapes the scope of the method in which it is raised, it becomes an output signal of the process that encodes the method, in order to be eventually propagated in the context of use of that method.

As an example, the encoding of the behavior ones in SIGNAL yields the following process. The clock of the input and output signals are synchronized to the input and output notification events. The process ones consists of one critical section in which the internal while loop is encoded by an over-sampling sub-process.

```
process ones = (? integer Inport; event start ! integer Outport; event done)
   (| start   ^= Inport
    | Outport := ocount when data=0
    | data    := Inport default rshift (data$1 init xnot(0))
    | ocount  := (ocount$1 init 0) + xand (data, 1)
    | done    ^= Outport
    |) where integer data, ocount;
end;
```

7 Putting polychrony to work for system design

The relational model of SIGNAL allows to accurately render the architecture of hardware devices at several levels of abstraction. For instance, one can reproduce the same behaviors as those found in conventional simulators :

- In an event-driven simulator, value changes produce events that can be scheduled either during the same physical time step (until each signal carries its last value for the current time (delta delays)) or at a future time step. Event-driven simulators are usually interpreted. They can handle asynchronous and timed devices, but are slow.
- In cycle-based simulators, events are synchronized w.r.t. a physical clock. Events are flattened and re-ordered (leveled) statically (i.e. at compile-time) in order to avoid reevaluating sections of code several times. The advantage of cycle-based simulators is generally speed, but only when signal activity is important. Otherwise, an evaluation which just reveals no change w.r.t. the previous one yields poor simulation speed compared to event-driven simulators.
- In a bus-transaction level simulator, processors that exchange data via a bus first need to have the bus granted. But, if the system is fully described at low-level, this can yield long simulation times, while the exchange could be considered at a higher level of abstraction to provide better simulation speed and simplified functional analysis.

7.1 Simulation techniques through model transformations

Synchronous languages focus on the chronology instead of the chronometry : this enables to reason directly on the signals and find a correct scheduling among them, as in cycle-based simulators.

In the same manner, the notion of over-sampling implemented in SIGNAL provides similar benefits as the event-driven simulation methods, with the notable exception that modeling in SIGNAL always yields compiled simulation code instead of requiring an interpretation mechanism by a run-time scheduler.

Access to physical time information is generally not handled. In SIGNAL, it is possible to generate a time calculation graph that is isomorphic to the functional graph of the simulated application. At each node of this graph, a calculation is made that takes the arrival date of each operand into account, takes the maximum of these values, and finally performs the addition of a value accounting for the delay of the current node operation.

In each case, the abstract signals handled by the compiler are tagged with a clock, that is basically another signal that controls the usefulness of the signal at the current cycle. That can be applied recursively : this tag is thus tagged by another tag. The tags are organized as a tree. The ordering among tags is rendered by nested conditionals in the generated C-code. This conditions the execution of code segments.

Several pitfalls should however be avoided when trying to fully benefit from this code generation technique : at the gate level for instance, a correct modeling of each gate needs to be found. In fact, in the zero-delay model, an acyclic logic cloud can be evaluated in a single step, and this can yield quite naturally to a purely monochronous operator (e.g. and, or, xor) that *constrains* the operand to be present at each clock cycle. The result of this modeling will be an *oblivious* compiled code simulator. To really compete with event-driven simulators for low-level activity circuits, this modeling needs to be tuned accordingly (by inserting as many $\perp$ as possible in the model). This is performed in several steps. Let us consider the example of a two inputs and-gate for simplicity :

- Each operand x and y is considered absent or present in the current cycle. If x is absent but y present, we use the cell operator that memorizes the previous value of a signal (i.e. x). We thus obtain : x'`:=`x `cell (event` y`)`. The same is applied to y.
- We then know that x' and y' are present at the same time and that we can apply the `and` synchronous operator on x' and y'.
- The output signal is generated only if it is different from previous cycle.

These models then act as filters on events (changes of values on the signals). Multiple data/control paths arise from this simple modeling. The same basic principles can be applied to other levels such as structural RTL level, or to translate VHDL code into SIGNAL equations and then to C code.

7.2 Polychrony and high-level synthesis

The concepts introduced in POLYCHRONY have direct applications to high-level synthesis.

From hierarchical conditional dependency graph to hardware The data-flow style of SIGNAL allows to generate hardware in a very simple way: each operator can be directly translated by a piece of hardware in which valuated input signals are processed, and produce output signals. As hardware devices always carry electrical values, the notion of absence/presence of signals should also be rendered. For this purpose, each signal is accompanied by a boolean signal that validates the presence of an informative event. Each operator thus produces both an output value and an output boolean tag.

However, a problem arises from this method: resources are not shared, while accurate guards information could be extracted from a global analysis and incorporated in the allocation and assignment phases. In fact the use of a compiler is compulsory : the method described above can lead to combinational loops that still causes problems in traditional EDA flows.

A hierarchical internal representation The data-flow graph built by the SIGNAL compiler represents the static assignment of signals and closely resembles to related compiler internal representation (e.g. [16]). Each node of the data-flow graph is guarded by a computed clock. This means that both the calculation nodes and the data dependencies (edges) are tagged by a clock computed by the compiler.

Because clocks are then hierarchized by the compiler, the data-flow graph is often presented as a hierarchical conditional dependency graph (HCDG). The hierarchy of clocks can be considered from the behavioral point of view (the hierarchization of clocks is applied through the whole graph). Clocks (or guards) are compared one against another and ordered with respect to their relative frequency.

This ordering enables one to discover mutual exclusions, which yield naturally to implementing notions of conditional resource sharing. Moreover, because of the consistent representations of control and data in the HCDG, this sharing technique can be generalized in a seamless way: registers shared by variables, functional units shared by calculi and ports shared by interfaces signals.

Transformation-based high-level synthesis The hierarchical conditional dependency graph allows to host design transformations without hovering from an intermediate representation to another: all transformations can directly be operated on the graph and rendered by a corresponding SIGNAL process in POLYCHRONY.

The HCDG allows to apply performance measurements during high-level synthesis (by operating design morphisms). These measurements have direct impact on the behavior of the hardware generated starting from the RTL code. For instance, registers sharing can take the lower-level combinational logic into account (as in [18]).

At a pre-synthesis level, the HCDG allows to determine the definition and use clocks of signals. Hence, dead-code elimination can easily be implemented by determining unused signals (i.e. of use-clock 0). The use of POLYCHRONY for pre-synthesis could further be investigated by considering the code motion techniques [13] consisting of the transformation of behavioral descriptions into either more lazy code (by encouraging the lazy evaluation of some instructions) or, conversely, by raising the execution probability of some nodes (in order to optimize the scheduling of the graph).

A synthesis-oriented refinement framework While most of the available synthesis tools focus on the synthesis of a single process at a time, POLYCHRONY allows for proceeding to local refinements: as soon as a sub-graph amenable to high-level synthesis is detected and synthesized into a datapath-controller pair, the compiler can naturally embed it into the initial specification. This requires using the notion of over-sampling, which enables the designer to produce a new simulator at a lower level of details without making major modifications of the initial graph. Part of the initial graph is then timed by the datapath-controller execution cycles and paced at a physical clock, but the remainder of the specification is unchanged. This technique is useful because it does not require the addition of buffers at the input-output interface of the sub-graph: during simulation, the control of the newly and automatically generated graph speeds up the execution of the synthesized sub-graph.

8 Conclusion

We have presented, reviewed and summarized the mathematical model and the formal methods present in the SIGNAL design tools to model and synthesize systems starting from early specification requirements down to sequential and distributed code generation.

Synchronous design models and languages provide intuitive (ontological) models for integrated circuits. In the relational mathematical model behind the design language SIGNAL, the affinity between circuits and programs goes beyond the domain of purely synchronous systems to embrace the context of integrated architectures consisting of synchronous circuits and desynchronization protocols: globally asynchronous and locally synchronous architectures.

The unique features of the relational model behind SIGNAL are to provide the notion of *polychrony*: the capability to describe circuits and systems with several clocks; and to support *refinement*: the ability to assist and support system design from the early stages of requirements specification to the later stages of synthesis and deployment.

The SIGNAL model provides a design methodology that spans from synchrony to asynchrony, from specification to implementation, from abstraction to concretization, from interfaces to implementations. SIGNAL gives the opportunity to seamlessly model circuits and devices at multiple levels of abstractions, by implementing

mechanisms found in many hardware simulators, while reasoning within a simple and formally defined mathematical model.

In the same manner, the flexibility inherent to the abstract notion of signal handled in the synchronous-desynchronized design model of SIGNAL invites and favors the design of correct by construction systems by means of well-defined transformations of system specifications (morphisms) that preserve the intended semantics and stated properties of the architecture under design.

1. AMAGBEGNON, T. P., BESNARD, L., LE GUERNIC, P. "Implementation of the dataflow synchronous language SIGNAL". In *Conference on Programming Language Design and Implementation*. ACM Press, 1995.
2. AUBRY, P. "Mises en oeuvre distribues de programmes synchrones". *Thèse de l'Universit de Rennes 1*. IFSIC, October 1997.
3. BENVENISTE, A., CAILLAUD B., AND LE GUERNIC, P. "Compositionality in dataflow synchronous languages: specification and distributed code generation". In *Information and Computation*, v. 163, pp. 125-171. Academic Press, 2000.
4. BENVENISTE, A., CASPI, P., LE GUERNIC, P., MARCHAND, H., TALPIN, J.-P., TRIPAKIS, S. "A protocol for loosely time-triggered architectures". In *Embedded Software Conference*. Springer Verlag, October 2002.
5. BENVENISTE, A., LE GUERNIC, P., JACQUEMOT, C. "Synchronous programming with events and relations: the SIGNAL language and its semantics". In *Science of Computer Programming*, v. 16, 1991.
6. BERRY, G., GONTHIER, G. "The ESTEREL synchronous programming language: design, semantics, implementation". In *Science of Computer Programming*, v. 19, 1992.
7. BERRY, G., RAMESH, S., SHYAMASUNDAR, R. K. "Communicating Reactive Processes". In *Symposium on Principles of Programming Languages*. ACM Press, 1993.
8. BERRY, G., SENTOVICH, E. "An Implementation of Constructive Synchronous Constructive Programs in Polis". In *Formal Methods in Systems Design 17(2)*. Kluwer Academic Publisher, 2000.
9. CARLONI, L. P., MCMILLAN, K. L., SANGIOVANNI-VINCENTELLI, A. L. "Latency-Insensitive Protocols". In *International Conference on Computer-Aided Verification*. Lecture notes in computer science v. 1633. Springer Verlag, July 1999.
10. CASPI, P. "Clocks in dataflow languages". In *Theoretical Computer Science*, v. 94. Elsevier, 1992.
11. CASPI, P., GIRAULT, A., JARD, C. "Distributed reactive systems". In *International Conference on Parallel and Distributed Computing Systems*. ISCA, 1994.
12. DE ALFARO, L., HENZINGER, T. A. "Interface theories for component-based design". In *International Workshop on Embedded Software*. Lecture Notes in Computer Science v. 2211, pp. 148-165. Springer-Verlag, 2001.
13. GUPTA, S., SAVIOU, N., DUTT, N., GUPTA, R., NICOLAU, A. "Speculation techniques for high-level synthesis of control-intensive designs". In *Design Automation Conference*. ACM Press, January 2001.
14. HALBWACHS, N. "Synchronous programming of reactive systems". Kluwer Academic Publishers, 1993.
15. HALBWACHS, N., CASPI, P., RAYMOND, P., PILAUD, D. "The synchronous data-flow programming language LUSTRE". In *Proceedings of the* IEEE, v. 79(9). IEEE Press, 1991.
16. JUAN, H.-P., CHAIYAKUL, V., GAJSKI, D. "Condition graphs for high-quality behavioral synthesis". In *International conference on computer-aided design*. IEEE Press, November 1994.

17. Kountouris, A., Wolinski, C. "Hierarchical conditional dependency graphs as a unifying design representation in the Codesis high-level synthesis system". In *International Symposium on System Synthesis*. Ieee Press, 2000.
18. Lakshminarayana, G., Raghunathan, A., Jha, N.K., Dey, S. "Transforming control-flow intensive designs to facilitate power management". In *International conference on computer-aided design*. Ieee Press, 1998.
19. Lee, E. A., Sangiovanni-Vincentelli, A. "A framework for comparing models of computation". In Ieee *transactions on computer-aided design*, v. 17, n. 12. Ieee Press, December 1998.
20. Marchand, H., Rutten, E., Le Borgne, M., Samaan, M. "Formal Verification of SIGNAL programs: Application to a Power Transformer Station Controller". In *Science of Computer Programming*, v. 41(1), pp. 85–104, 2001.
21. Nowak, D., Talpin, J.-P., Le Guernic, P. "Synchronous structures". In *International Conference on Concurrency Theory*. Springer Verlag, August 1999.