An algebraic theory for behavioral modeling and protocol synthesis in system design

Jean-Pierre Talpin and Paul Le Guernic
INRIA-IRISA, Campus de Beaulieu, 35042 Rennes, France

Abstract. The design productivity gap has been recognized by the semiconductor industry as one of the major threats to the continued growth of system-on-chips and embedded systems. Ad-hoc system-level design methodologies, that lifts modeling to higher levels of abstraction, and the concept of intellectual property (IP), that promotes reuse of existing components, are essential steps to manage design complexity. However, the issue of compositional correctness arises with these steps. Given components from different manufacturers, designed with heterogeneous models, at different levels of abstraction, assembling them in a correct-by-construction manner is a difficult challenge. We address this challenge by proposing a process algebraic model to support system design with a formal model of computation and serve as a type system to capture the behavior of system components at the interface level. The proposed algebra is conceptually minimal, equipped with a formal semantics defined in a synchronous model of computation. It supports a scalable notion and a flexible degree of abstraction. We demonstrate its benefits by considering the type-based synthesis of latency-insensitive protocols, showing that the synthesis of component wrappers can be optimized by behavioral information carried by interface type descriptions and yield minimized stalls and maximized throughput.

Keywords: Synchrony, process algebra, compositional modeling, GALS design.

1. Introduction

The design productivity gap has been recognized by the semiconductor industry as one of the major threats to the continued growth of complex system-chips and their applications. System level design methodologies that lift design methods at higher-level of abstraction are essential to manage design complexity. A number of advances in high-level modeling and validation have been proposed over the past decade in an attempt to improve the level of abstraction in system design, most of these enable greater reuse of existing intellectual property (IP) blocks.

Design correctness is an important step in this move. Given the complexity of system-level designs, it is important that the composition of system-level IP blocks be guaranteed correct. However, *a posteriori* validation of component compositions is a difficult problem. Techniques are needed that ensure design correctness as a part of the design process itself. To address this issue, methodological precepts have been developed that separately focus on design reuse and correctness by

construction. Both reuse and elevation of abstraction critically depend on guaranteed design correctness. To improve the state of the art in component composition from existing component libraries, we specifically seek to address the following issue: given a high level architectural description and a library of implemented components, how can one automate the selection of implementation of virtual components from the library, and automatically ensure composability ?

Our approach is based on a high-level modeling and specification methodology that ensures compositional correctness through an algebra capturing behavioral aspects of component interfaces. It is presented via a minimalist algebra, called the iSTS (implicit and synchronous transition systems), that is akin to Pnueli's synchronous transition systems (STS, [14]) and Dijkstra's guarded commands [7]. This minimalist formalism allows us to specify the state transitions, synchronization relations and scheduling constraints implied by a given system, in the presence of multiple-clocked synchrony (i.e. polychrony). Our approach consists of using this formalism as a type system to describe the behavior of system components and allow for global model transformations to be performed on the system based on behavioral type information. This approach builds upon previous work on scalable design exploration using refinement/abstraction-based design methodologies [15], implemented in the Polychrony workbench [16]. It is both conceptually minimal and equipped with a formal semantics defined in a multi-clocked synchronous model of computation [12].

Roadmap After a detailed and informal exposition of our formalism, Section 2, and of its model of computation, Section 3, we exemplify its use as a behavioral type system in system design by the definition of a model transformations, Section 6, and formal methodologies, Section 4, applied to the synthesis of latency-insensitive protocol, Section 6.

2. A polychronous algebraic notation

We start with an informal outline of an algebraic formalism which we call the iSTS (implicit synchronous transition systems). The key notions put to work in this notation are essentially borrowed to Pnueli's STS [14] and Dijkstra's guarded command language [7]. In the iSTS, a process consists of simultaneous propositions that manipulate signals. A signal is an infinite flow of values that is sampled by a discrete series of instants or reactions. An event corresponds to the value carried by a signal during a particular reaction or instant. The main features of the iSTS notation are put together in the process P, below, that

describes the behavior of a counter modulo 2, noted P, through a set of simultaneous *propositions*, labeled from (1) to (3):

$$\begin{array}{lcll} & & \neg s^0 & (1) \\ P & \stackrel{\text{def}}{=} & \mid \hat{x} \Rightarrow s' = \neg s & (2) \\ & & \mid s \Rightarrow \hat{y} & (3) \end{array}$$

- Proposition (1) is an invariant. It says that the initial value of the signal s, denoted by s^0, is false. This is written $\neg s^0$.
- Proposition (2) is a guarded command. It says that if the signal x is present during a given reaction then the value of s is toggled.
  - The leftmost part of the sentence, the proposition $\hat{x}$, is a condition or a guard. It denotes the *clock* of x. It is true if the signal x is present during the current reaction.
  - The rightmost part of the sentence, the proposition $s' = \neg s$, is a transition. The term s' refers to the next value of the signal s. The proposition $s' = \neg s$ says that the next value of s is the negation of the present value of s.
- Proposition (3) says "if s is true then y is present".

Formal syntax The formal syntax of the iSTS is defined by the inductive grammar P in Table I. A process P manipulates boolean values noted $v \in \{0, 1\}$ and signals noted x, y, z. A location l refers to the initial value x^0, the present value x and the next value x' of a signal. A reference r stands for either a value v or a signal x.

A clock expression e is a proposition on boolean values. When true, a clock e defines a particular period in time. The clocks 0 and 1 denote events that never/always happen. The clock $x = r$ denotes the proposition: "x is present and holds the value r". Particular cases are:
- the clock noted $\hat{x} \stackrel{\text{def}}{=} (x = x)$ means that "x is present"
- the clock noted $x \stackrel{\text{def}}{=} (x = 1)$ means that "x is true"
- the clock noted $\neg x \stackrel{\text{def}}{=} (x = 0)$ means that "x is false"

Clocks are propositions combined using the logical combinators of conjunction $e \wedge f$, to mean that both e and f hold, disjunction $e \vee f$, to mean that either e or f holds, and symmetric difference $e \setminus f$, to mean that e holds and not f. A process P consists of the simultaneous composition of elementary propositions. 1 is the process that does nothing. The proposition $l = r$ means that "l holds the value r" [1]. The proposition $x \rightarrow l$ means that "l cannot happen before x". The

[1] In Section 1, we wrote $x' = \neg x$ to mean $x = 1 \Rightarrow x' = 0 \,|\, x = 0 \Rightarrow x' = 1$.

process $e \Rightarrow P$ is a guarded command. It means: "if e is present then P holds". Processes are combined using synchronous composition $P \,|\, Q$ to denote the simultaneity of the propositions P and Q. Restricting a signal name x to the lexical scope of a process P is written P/x. We refer to the *free variables* $\mathrm{vars}(P)$ of a process P as the set of signal names that occur free in the lexical scope of P.

Table I. Formal syntax of iSTS algebra

$$\begin{array}{llll} \text{(reference)} & r ::= x \,|\, v & \text{(clock)} & e, f ::= 0 \,|\, x = r \,|\, e \wedge f \,|\, e \vee f \,|\, e \setminus f \,|\, 1 \\ \text{(location)} & l ::= x^0 \,|\, x \,|\, x' & \text{(process)} & P, Q ::= 1 \,|\, l = r \,|\, x \rightarrow l \,|\, e \Rightarrow P \,|\, (P \,|\, Q) \,|\, P/x \end{array}$$

Notational conventions In the formal presentation of the iSTS, we restrict ourself to a subset of the elementary propositions in the grammar of Table I, which we call atoms a. Conversely, syntactic shortcuts used in the examples are defined as follows:

$$\text{(atoms)} \quad a, b ::= x^0 = v \,|\, l = y \,|\, x \rightarrow l \text{ where } l ::= x \,|\, x'$$

$$\begin{array}{ll} l = v \stackrel{\text{def}}{=} (l = x \,|\, x^0 = v \,|\, x' = x)/x \text{ iff } x \neq l \neq x' & \hat{x} \stackrel{\text{def}}{=} (x = x) \\ l := x \stackrel{\text{def}}{=} (l = x \,|\, x \rightarrow l) & l \stackrel{\text{def}}{=} (l = 1) \\ \hat{x} = \hat{y} \stackrel{\text{def}}{=} (\hat{x} \Rightarrow \hat{y} \,|\, \hat{y} \Rightarrow \hat{x}) & \neg l \stackrel{\text{def}}{=} (l = 0) \end{array}$$

3. A polychronous model of computation

To deal with the apparent heterogeneity of synchrony and asynchrony in GALS architectures, designers usually consider stratified models, such as CSP (communicating sequential processes) or Kahn networks.

By contrast, polychrony (multi-clocked synchrony) establishes a continuum from synchrony to asynchrony: modeling, design, transformation, verification issues are captured within the same model and hence independently of spatial and temporal considerations implied by a synchronous (local) or an asynchronous (global) viewpoint. The definition of uniform methodologies for the formal design of GALS architectures has been the subject of recent and detailed studies in [12] and [15].

In this section, we cast polychrony in the context of the iSTS algebra to explore abstraction and refinement relations between system-level models in a way geared towards the aim of compositional system design. The polychronous model of computation, inspired from [11] and proposed in [12], consists of a *domain* of traces that does not differentiate synchrony from asynchrony and of semi-lattice structures that render synchrony and asynchrony using specific timing equivalence relations.

3.1. Domain of polychrony

We consider a partially-ordered set $(\mathcal{T}, \leq, 0)$ of tags. A tag $t \in \mathcal{T}$ denotes a symbolic instant or a period in time. We note $C \in \mathcal{C}$ a *chain* of $\mathcal{T}$. Signals, behaviors and processes are defined starting as follows:

DEFINITION 1 (polychrony).
- *An* event $e \in \mathcal{T} \times \mathcal{V}$ *is the pair of a tag and a value.*
- *A signal* $s \in C \to \mathcal{V}$ *is a function from a* chain *of tags to values.*
- *A* behavior $b \in \mathcal{B}$ *is a function from names* $x \in \mathcal{X}$ *to signals* $s \in \mathcal{S}$.
- *A* process $p \in \mathcal{P}$ *is a set of behaviors that have the same domain.*

We write $\mathrm{tags}(s)$ *and* $\mathrm{tags}(b)$*for the tags of a signal* s *and of a behavior* b*;* $b|_X$ *for the projection of a behavior* b *on* $X \subset \mathcal{X}$ *and* $b/X = b|_{\mathrm{vars}(b)\setminus X}$ *for its complementary;* $\mathrm{vars}(b)$ *and* $\mathrm{vars}(p)$ *for the domains of* b *and* p.

Synchronous composition $p \,\|\, q$ of two processes p and q is defined by the union of all behaviors b (from p) and c (from q) which are synchronous: all signals along the interface $I = \mathrm{vars}(p) \cap \mathrm{vars}(q)$ between p and q carry the same values at the same time tags.

$$p \,\|\, q = \{ b \cup c \mid (b, c) \in p \times q, I = \mathrm{vars}(p) \cap \mathrm{vars}(q), b|_I = c|_I \}$$

Scheduling structure To render scheduling relations between events occurring at the same time tag t, we equip the domain of polychrony with a scheduling relation, noted $t_x \to t'_y$, defined on the domain of dates $\mathcal{D} = \mathcal{T} \times \mathcal{X}$, to mean that the event along the signal named y at t' may not happen before x at t. When no ambiguity is possible on the identity of b in a scheduling constraint, we write it $t_x \to t_y$. We constraint scheduling $\to$ to contain causality so that $t < t'$ implies $t_x \to^b t'_x$ and $t_x \to^b t'_x$ implies $\neg(t' < t)$.

3.2. Synchronous structure of polychrony

Building upon this domain of polychrony defined in Section 3.1, we define the semi-lattice structure which relationally denotes synchronous behaviors in this domain. The intuition behind this relation is to consider a signal as an elastic with ordered marks on it (tags). If the elastic is stretched, marks remain in the same relative and partial order but have more space (time) between each other. The same holds for a set of elastics: a behavior. If they are equally stretched, the order between marks is unchanged.

DEFINITION 2 (clock equivalence [12]). *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 *on* $\mathcal{T}$ *which satisfies*

$$\left|\begin{array}{l}\forall t,t' \in \text{tags}(b), t \leq f(t) \wedge (t < t' \Leftrightarrow f(t) < f(t')) \\ \forall x,y \in \text{vars}(b), \forall t \in \text{tags}(b(x)), \forall t' \in \text{tags}(b(y)), t_x \rightarrow^b t'_y \Leftrightarrow f(t)_x \rightarrow^c f(t')_y \\ \forall x \in \text{vars}(b), \text{tags}(c(x)) = f(\text{tags}(b(x))) \wedge \forall t \in \text{tags}(b(x)), b(x)(t) = c(x)(f(t))\end{array}\right.$$

Stretching is a partial-order relation and defines clock equivalence: b and c are clock-equivalent*, written $b \sim c$, iff there exists d s.t. $c \geq d \leq b$.*

3.3. Asynchronous structure of polychrony

The asynchronous structure of polychrony is modeled by weakening the clock-equivalence relation to allow for comparing behaviors w.r.t. the sequences of values signals hold regardless of the time at which they hold these values. *Relaxation* individually stretches the signals of a behavior in a way preserving scheduling constraints. It is a partial-order that defines flow-equivalence: two behaviors are flow-equivalent iff their signals hold the same values in the same order.

DEFINITION 3 (flow equivalence [12]). *A behavior c is a* relaxation *of b, written $b \sqsubseteq c$, iff* $\text{vars}(b) = \text{vars}(c)$ *and, for all $x \in \text{vars}(b)$, there exists a bijection f_x on $\mathcal{T}$ which satisfies*

$$\left|\begin{array}{l}\forall t,t' \in \text{tags}(b(x)), t \leq f_x(t) \wedge t < t' \Leftrightarrow f_x(t) < f_x(t')) \\ \forall t \in \text{tags}(b(x)), \forall t' \in \text{tags}(b(y)), t_x \rightarrow^b t'_y \Leftrightarrow (f_x(t))_x \rightarrow^c (f_y(t'))_y \\ \text{tags}(c(x)) = f_x(\text{tags}(b(x))) \wedge \forall t \in \text{tags}(b(x)), b(x)(t) = c(x)(f_x(t))\end{array}\right.$$

b, c are flow-equivalent*, written $b \approx c$, iff there exists d s.t. $b \sqsupseteq d \sqsubseteq c$.*

Asynchronous composition $p \parallel q$ is defined by considering the partial-order structure induced by the relaxation relation . The parallel composition of p and q consists of behaviors d that are relaxations of behaviors b and c from p and q along shared signals $I = \text{vars}(p) \cap \text{vars}(q)$ and that are stretching of b and c along independent signals of p and q.

$$p \parallel q = \left\{ d \in \mathcal{B}|_{\text{vars}(p) \cup \text{vars}(q)} \mid \exists (b,c) \in p \times q, d/I \geq (b/I \,\|\, c/I) \wedge b|_I \sqsubseteq d|_I \sqsupseteq c|_I \right\}$$

3.4. From synchronous to asynchronous structures

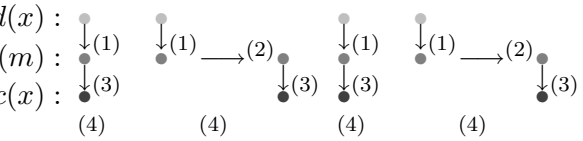
Polychrony is designed to render the synchronous hypothesis and relate it to asynchronous architectures using communication with unbounded delay. In an embedded architecture, however, the flow of a signal usually slides from another as the result of introducing finite delays using, e.g., finite Fifo buffers or *relay stations*. Definition 4 formalizes this relation by considering the timing deformation between an initial behavior b and a final behavior c performed by a one-place FIFO buffer of internal

signal m and behavior d. The behavior d is defined by stretching $b \leq d/m$ and c/x by d/mx. Let us write $\mathrm{pred}_C(t)$ (resp. $\mathrm{succ}_C(t)$) for the immediate predecessor (resp. successor) of the tag t in the chain C.

DEFINITION 4 (finite relaxation). *The behavior c is a 1-relaxation of x in b, written $b \sqsubseteq_1^x c$ iff* $\mathrm{vars}(b) = \mathrm{vars}(c)$ *and there exists a signal m, a behavior d and a chain $C = \mathrm{tags}(d(m)) = \mathrm{tags}(d(x)) \cup \mathrm{tags}(c(x))$ such that $d/m \geq b$, $d/mx = c/x$ and, for all $t \in C$,*

$$
\begin{array}{lll}
(1) & t \in \mathrm{tags}(d(x)) & \Rightarrow\ d(x)(t) = d(m)(t) \wedge t_x \rightarrow_d t_m \\
(2) & t \notin \mathrm{tags}(d(x)) & \Rightarrow\ d(m)(t) = d(m)(\mathrm{pred}_C(t)) \\
(3) & t \in \mathrm{tags}(c(x)) & \Rightarrow\ c(x)(t) = d(m)(t) \wedge \forall y \in \mathrm{vars}(d) \setminus m, t_y \rightarrow_d^* t_x \\
(4) & t \in \mathrm{tags}(c(x)) & \Rightarrow\ c(x)(t) = d(x)(t) \vee c(x)(\mathrm{succ}_C(t)) = d(x)(t)
\end{array}
$$

Rule (1) says that, when an input $d(x)$ is present at some time t, then $d(m)$ takes its value. If no input is present along x at t, rule (2), then $d(m)$ takes its previous value. Rule (3) says that, if the output $c(x)$ is present at t, then it is defined by $d(m)(t)$. Finally, rule (4) requires this value to either be the present or previous value of the input signal $d(x)$, binding the size of the buffer to one place.



Definition 4 accounts for the behavior of bounded FIFOs in a way that preserves scheduling relations. It implies a series of (reflexive-anti-symmetric) relations $\sqsubseteq_n$ (for $n > 0$) which yields the (series of) reflexive-symmetric flow relations $\approx_n$ to identify processes of same flows up to a flow-preserving first-in-first-out buffer of size n. We write $b \sqsubseteq_1 c$ iff $b \sqsubseteq_1^x c$ for all $x \in \mathrm{vars}(b)$, and, for all $n > 0$, $b \sqsubseteq_{n+1} c$ iff there exists d such that $b \sqsubseteq_1 d \sqsubseteq_n c$. The largest equivalence relation modeled in the polychronous model of computation consists of behaviors equal up to a timing deformation performed by a finite FIFO protocol.

DEFINITION 5. *b and c are finite flow-equivalent, written $b \approx^* c$, iff there exists $0 \leq N < \infty$ s.t. $b \approx_N c$.*

3.5. Denotational semantics of the algebra

The detailed presentation and extension of the polychronous model of computation allows us to give a denotational model to the iSTS notation introduced in Section 2. This model consists of relating a proposition P to the set of behaviors p it denotes.

Meaning of clocks. Let us start with the denotation of a clock expression e (Table II). The meaning $[\![e]\!]_b$ of a clock e is defined relatively to a given behavior b and consists of the set of tags satisfied by the proposition e in the behavior b. In Table II, the meaning of the clock $x = v$ (resp. $x = y$) in b is the set of tags $t \in \text{tags}(b(x))$ (resp. $t \in \text{tags}(b(x)) \cap \text{tags}(b(y))$) such that $b(x)(t) = v$ (resp. $b(x)(t = b(y)(t))$. In particular, $[\![\hat{x}]\!]_b = \text{tags}(b(x))$. The meaning of a conjunction $e \wedge f$ (resp. disjunction $e \vee f$ and difference $e \setminus f$) is the intersection (resp. union and difference) of the meaning of e and f. Clock 0 has no tags.

Table II. Denotational semantics of clock expressions

$$
\begin{array}{ll}
[\![1]\!]_b=\text{tags}(b) \quad [\![0]\!]_b = \emptyset & [\![e \wedge f]\!]_b=[\![e]\!]_b \cap [\![f]\!]_b \\
[\![x = v]\!]_b=\{t \in \text{tags}(b(x)) \,|\, b(x)(t) = v\} & [\![e \vee f]\!]_b=[\![e]\!]_b \cup [\![f]\!]_b \\
[\![x = y]\!]_b=\{t \in \text{tags}(b(x)) \cap \text{tags}(b(y)) \,|\, b(x)(t) = b(y)(t)\} & [\![e \setminus f]\!]_b=b[\![e]\!]_b \setminus [\![f]\!]_b
\end{array}
$$

Meaning of propositions. The denotation of a clock expression by a set of tags yields the denotational semantics of propositions P, written $[\![P]\!]$, Table III. The meaning $[\![P]\!]^e$ of a proposition P is defined with respect to a clock expression e. Where this information is absent, we assume $[\![P]\!] = [\![P]\!]^1$ to mean that P is an invariant (and is hence independent of a particular clock). The meaning of an initialization $x^0 = v$, written $[\![x^0 = v]\!]^e$, consists of all behaviors defined on x, written $b \in \mathcal{B}|_x$ such that the initial value of the signal $b(x)$ equals v. Notice that it is independent from the clock expression e provided by the context. In Table III, we write $\mathcal{B}|_X$ for the set of all behaviors of domain X, $\min(C)$ for the minimum of the chain of tags C, $\text{succ}_t(C)$ for the immediate successor of t in the chain C and $\text{vars}(P)$ and $\text{vars}(e)$ for the set of free signal names of P and e.

Table III. Denotational semantics of propositions

$$
\begin{array}{l}
[\![x = y]\!]^e=\{b \in \mathcal{B}|_{\text{vars}(e)\cup\{x,y\}} \,|\, \forall t \in [\![e]\!]_b, \\
\qquad t \in \text{tags}(b(x)) \wedge t \in \text{tags}(b(y)) \wedge b(x)(t) = b(y)(t)\} \\
[\![y \to x]\!]^e=\{b \in \mathcal{B}|_{\text{vars}(e)\cup\{x,y\}} \,|\, \forall t \in [\![e]\!]_b, \\
\qquad t \in \text{tags}(b(x)) \Rightarrow t \in \text{tags}(b(y)) \wedge t_y \to^b t_x\} \\
[\![x' = y]\!]^e=\{b \in \mathcal{B}|_{\text{vars}(e)\cup\{x,y\}} \,|\, \forall t \in [\![e]\!]_b, \\
\qquad t \in C = \text{tags}(b(x)) \wedge t \in \text{tags}(b(y)) \wedge b(x)(\text{succ}_t(C)) = b(y)(t)\} \\
[\![y \to x']\!]^e=\{b \in \mathcal{B}|_{\text{vars}(e)\cup\{x,y\}} \,|\, \forall t \in [\![e]\!]_b, \\
\qquad t \in C = \text{tags}(b(x)) \Rightarrow t \in \text{tags}(b(y)) \wedge t_y \to^b (\text{succ}_t(C))_x\} \\
[\![x^0 = v]\!]^e=\{b \in \mathcal{B}|_x \,|\, b(x)(\min(\text{tags}(b(x)))) = v\} \\
[\![f \Rightarrow P]\!]^e=[\![P]\!]^{e\wedge f} \quad [\![P \,|\, Q]\!]^e = [\![P]\!]^e \,|\, [\![Q]\!]^e \quad [\![P/x]\!]^e = \{c \leq b/x \,|\, b \in [\![P]\!]^e\}
\end{array}
$$

A proposition $x = y$ at clock e is denoted by behaviors b defined on $\text{vars}(e) \cup \{x, y\}$ and s.t. all tags of $t \in [\![e]\!]_b$ belong to $b(x)$ and to $b(y)$ and hold the same value. A scheduling specification $y \rightarrow x$ at clock e denotes the behaviors b on $\text{vars}(e) \cup \{x, y\}$ which, for all tags $t \in [\![e]\!]_b$, requires x to preceed y: if t is in $b(x)$ then it is necessarily in $b(y)$ and satisfies $t_y \rightarrow^b t_x$. The propositions $x' = y$ and $y \rightarrow x'$ is interpreted similarly by considering the tag t' that is the successor of t in the chain C of x. The behavior of a guarded command $f \Rightarrow P$ at the clock e is equal to the behavior of P at the clock $e \wedge f$. The meaning of a restriction P/x consists of the behaviors c of which a behavior b/x from P are a stretching of. The behavior of $P \,|\, Q$ consists of the composition of the behaviors of P and Q.

3.6. Properties of the denotational semantics

A remarkable syntactic property of iSTS with respect to the semi-lattice of clock equivalence is that, if clock equivalence holds between two processes P and Q, then it also holds in any context formed by "pluging" either of the processes P or Q to a term $K[] ::= [] \,|\, (P \,|\, K[]) \,|\, (e \Rightarrow K[]) \,|\, (K[]/x)$. From the point of view of compositional design, this property is important. It allows one to rely on the behavior expressed once and for all by a process P to represent a class of processes equivalent by clock equivalence and in any possible context K.

PROPERTY 1 (congruence). *If $P \sim Q$ then $K[P] \sim K[Q]$.*

Closure Property 2 states that the semantics of the iSTS is closed by clock-equivalence. It means that, whenever a process P has a given behavior $b \in [\![P]\!]$, then it also has any behavior c that is a stretch of b. This is an important assessment on the expressive capability of the iSTS: the property of closure implies that, whenever the intent of a given proposition is to model a particular scenario, a particular invariant of a system, then its implicit extent is the class of all behaviors that are clock-equivalent to this scenario.

PROPERTY 2 (closure). *If $b \in [\![P]\!]$ and $b \leq c$ then $c \in [\![P]\!]$.*

Expressibility Property 3 extends this result to finite flow equivalence with the aim of showing that the semantics of iSTS is expressive enough to model systems composed of synchronous processes communicating via reliable and bounded protocols, hence, encompass the specification of GALS architectures. It says that, whenever a given process or system has a given behavior b, and whenever c is a finite relaxation of b, then another process Q can be found such that c is a behavior of $P \,|\, Q$,

demonstrating the closure of the iSTS for finite flow equivalent behaviors. Property 3 can also be seen as a justification on the lack of an operator of parallel composition in the iSTS. Such an operator would model an unreliable (because unbounded) communication medium.

PROPERTY 3 (expressibility).
If $b \in [\![P]\!]$ *and* $b \sqsubseteq^* c$ *then there exists* Q *s.t.* $c \in [\![P \,|\, Q]\!]$.

Finding a process Q satisfying Property 3 amounts to consider the finite FIFO buffer protocols that are directly implied by the definition of the relation of relaxation. The model of a 1-place FIFO buffer from x to y is noted $\mathsf{fifo}_1^{x,y}$, below. It uses two local signals r and s. The initial value of the state variable s is set to true (proposition $s^0 = 1$). The state variable s and the register are synchronized (proposition $\hat{s} = \hat{r}$). When s is true, the FIFO awaits the presence of the input signal x (proposition $s = \hat{x}$) and defines the next value of the register r by the value of x (proposition $s \Rightarrow r' := x$). When s is false, it synchronizes with output signal y (proposition $\neg s = \hat{y}$), sends the current value of the register r (proposition $\neg s \Rightarrow y := r$) and sustains the value of r (proposition $r' := r$). If either a value is loaded or stored then the status of s changes (proposition $\hat{s} \Rightarrow s' = \neg s$).

$$\mathsf{fifo}_1^{x,y} \stackrel{\text{def}}{=} \left[\begin{pmatrix} s^0 = 1 \\ \hat{s} = \hat{r} \end{pmatrix} \mid \begin{pmatrix} s = \hat{x} \\ s \Rightarrow r' := x \end{pmatrix} \mid \begin{pmatrix} \neg s = \hat{y} \\ \neg s \Rightarrow y := r \end{pmatrix} \mid \begin{pmatrix} \neg s \Rightarrow r' := r \\ \hat{s} \Rightarrow s' := \neg s \end{pmatrix} \right] /rs$$

Compared to the input signal x, one easily observe that the output signal y of the protocol $\mathsf{fifo}_1^{x,y}$ has all tags shifted by at most one place (the shift cannot pass the next input). This yields the result expected in Property 3. Starting from the protocol $\mathsf{fifo}_1^{x,y}$, finite FIFO buffers of length $N > 1$ and of width $M > 1$ can easily be defined by induction on N and M. To attach a finite FIFO buffer to a given process P, we define the template or generic process $\mathsf{fifo}_N\langle P \rangle$ to redirect the input signals $x_{1..m} = \mathrm{in}(P)$ and the output signals $x_{m+1..n} = \mathrm{vars}(P) \setminus \mathrm{in}(P)$ of P to an input buffer $\mathsf{fifo}_N^{x_{1..m},y_{1..m}}$ and an output buffer $\mathsf{fifo}_N^{y_{m+1..n},x_{m+1..n}}$. This is done by substituting the free variables $x_{1..n}$ by fresh signal names $y_{1..n}$. The definition of the function $\mathrm{in}(P)$ is given in Appendix.

$$\begin{aligned}
\mathsf{fifo}_N^{x,y} &\stackrel{\text{def}}{=} \left(\mathsf{fifo}_1^{x,z} \,|\, \mathsf{fifo}_{N-1}^{z,y} \right) /z, \ \forall M, N > 1 \\
\mathsf{fifo}_N^{x_{1..m},y_{1..m}} &\stackrel{\text{def}}{=} \left(\mathsf{fifo}_N^{x_{1..m-1},y_{1..m-1}} \,|\, \mathsf{fifo}^{x_m,y_m} \right) \\
\mathsf{fifo}_N\langle P \rangle &\stackrel{\text{def}}{=} \left(\mathsf{fifo}_n^{x_{1..m},y_{1..m}} \,|\, (P[y_i/x_i]_{i=1}^n) \,|\, \mathsf{fifo}_N^{y_{m+1..n},x_{m+1..n}} \right) /y_{1..n}
\end{aligned}$$

4. From processes to behavioral types

In the terminology of [8], a process P defines the *dynamic interface* of a given system: it defines a transition system that specifies the invariants of its evolution in time. The abstraction of a process P by its *static interface* $\hat{P}$ is defined by the type inference system $e \vdash P : Q$. To a process P in a context of clock e, it associates the clock and scheduling relations Q that form a static abstraction of P. We assume the scheduling relation to be transitive: $e \Rightarrow x \rightarrow y \,|\, f \Rightarrow y \rightarrow z$ implies $(e \wedge f) \Rightarrow x \rightarrow z$, and distributive: $e \Rightarrow x \rightarrow y \,|\, f \Rightarrow x \rightarrow y$ implies $(e \vee f) \Rightarrow x \rightarrow y$).

$$\vdash 1 : 1 \qquad \vdash x^0 = v : 1 \qquad e \vdash x \rightarrow y : e \Rightarrow x \rightarrow y$$

$$e \vdash x = y : e \Rightarrow x = y \qquad e \vdash x' = y : (e \Rightarrow (\hat{x} = \hat{y})) \,|\, (e \Rightarrow \hat{x}) \,|\, (e \Rightarrow \hat{y})$$

$$\frac{e \wedge f \vdash P : Q}{e \vdash f \Rightarrow P : Q} \qquad \frac{e \vdash P : Q}{e \vdash P/x : Q/x} \qquad \frac{e \vdash P_1 : Q_1 \quad e \vdash P_2 : Q_2}{e \vdash P_1 \,|\, P_2 : Q_1 \,|\, Q_2}$$

The correctness of the inference system is stated by a denotational containment relation, i.e., a soundness property:

PROPERTY 4. *$[\![P]\!] \subseteq [\![\hat{P}]\!]$ and, if $e \vdash P : Q$, then $[\![P]\!]^e \subseteq [\![Q]\!]^e$.*

4.1. Hierarchization

Behavioral type inference associates a system of clock equations $\hat{P}$ with an iSTS process P. Based on this information, a canonical representation of the partial order between the signal clocks of a process P can be determined by a transformation called *hierarchization* [1]. The control-flow graph it constructs is the medium which we use to perform the verification, transformation and design exploration.

EXAMPLE 1. *The implications of hierarchization to code generation can be outlined by considering the specification of a one-place buffer. The process* buffer *has input x, output y and implements two functionalities. One is the process* alternate *which desynchronizes the signals x and y by synchronizing them to the true and false values of an alternating boolean signal* b*. The other functionality is the process* current*. It defines a cell in which values are stored at the input clock $\hat{x}$ and loaded at the output clock $\hat{y}$.*

$$\begin{aligned}
\mathsf{buffer} < x, y > &\stackrel{\text{def}}{=} \mathsf{alternate} < x, y > \,|\, \mathsf{current} < x, y > \\
\mathsf{alternate} < x, y > &\stackrel{\text{def}}{=} (s^0 = 1 \,|\, \hat{x} = s \,|\, \hat{y} = \neg s \,|\, s' := \neg s) \\
\mathsf{current} < x, y, b > &\stackrel{\text{def}}{=} (r^0 = b \,|\, r' := x \,|\, \hat{x} \Rightarrow y := x \,|\, \hat{y} \setminus \hat{x} \Rightarrow y := r)
\end{aligned}$$

We observe that s defines the master clock of buffer. *There are two other synchronization classes, x and y, that corresponds to the true and false values of the boolean flip-flop variable s, respectively. This defines three nodes in the control-flow graph of the generated code. At the master clock $\hat{s}$, the value of s is calculated from zs, its previous value. At the sub-clock $s = \hat{x}$, the input signal x is read. At the sub-clock $\neg s = \hat{y}$ the output signal y is written. Finally, the new value of zs is determined.*

```
buffer_iterate () {
  s = !zs;
  cy = !s;
  if (s && !r_buffer_i(&x)) return FALSE;
  if (cy) { y = x; w_buffer_o(y); }
  zs = s;
  return TRUE; }
```

Figure 1. C code generated for the one-place buffer specification.

In Example 1, we observe that hierarchization involves both free and bound signals of a proposition P. Hence, Definition 6 considers the lift $\overline{P}$ of bound variables in P defined by induction on P by:
- $\overline{a/X} = a/X$ and $\overline{P/x} = Q/X \cup \{x\}$ iff $\overline{P} = Q/X$
- $\overline{e \Rightarrow P} = (e \Rightarrow Q)/X$ iff $\overline{P} = Q/X$ and $\mathrm{vars}(e) \cap X = \emptyset$
- $\overline{P_1 \,|\, Q_1} = (P_2 \,|\, Q_2)/XY$ iff $\overline{P}_1 = P_2/X$ and $\overline{Q}_1 = Q_2/Y$

Definition 6 uses the static abstraction $\hat{Q}$ of $\overline{P} = Q/X$ to find clock equivalence classes. In Definition 6, we write $P \models Q$ iff the proposition P implies the proposition Q.

DEFINITION 6 (hierarchization). *The* clock relation $\preceq_P$ *of a process P is the partial order relation between elementary clocks $(x = r)$, written h, and defined with $Q = \overline{P}$ by -1. if $\hat{Q} \models \hat{x} = h$ or $\hat{Q} \models (x = v) = h$ then $\hat{x} \preceq h$ -2. if $\hat{x} \preceq h_1$, $\hat{x} \preceq h_2$ and $\hat{Q} \models h = h_1 \star h_2$ for any $\star \in \{\wedge, \vee, \backslash\}$ then $\hat{x} \preceq h$. We say that h_1 and h_2 are clock-equivalent, written $h_1 \precdot\!\!\cdot h_2$, iff $h_1 \preceq h_2$ and $h_2 \preceq h_1$.*

Rule 1 defines equivalence classes for synchronous signals and constructs elementary partial orders for $(x = v)$ and $\hat{x}$. Rule 2 connects partial orders. The insertion algorithm, introduced in [1], yields a canonical representation by observing that there exists a unique minimum clock below x such that rule 2 holds. A process whose clock relation is hierarchic, in the sense of Definition 7, meets a necessary criterion for being implemented by a sequential program.

DEFINITION 7. *A process P is* hierarchic *iff $\preceq_P$ has a minimum.*

4.2. Transformation of hierarchical processes

On the path towards a sequential implementation of a hierarchical process, an important step is the structural transformation of a process P into a model from which formal design properties can easily be checked and target code easily and structurally generated. We outline such transformations implemented in the Polychrony workbench [16], They consist of representing a process P either into a disjunctive form, noted $\mathcal{D}_P$, or into a hierarchical form, noted $\mathcal{H}_P$. Table IV examplifies such a transformation using the counter modulo 2 of Example 1.

The disjunctive form $\mathcal{D}_P$ of the process P consists of its representation as the composition of elementary guards $g \Rightarrow a$ consisting of a conjunctive clock proposition g $[g ::= 0 \,|\, 1 \,|\, (x = r) \,|\, g \wedge g \,|\, g \setminus g]$ and of an atomic action a. The hierarchical form $\mathcal{H}_P$ of the process P consists of representing it according to the tree-like structure of its associated clock relation $\preceq_P$. One of the benefits of this representation is to make the control-flow of the process explicit to facilitate code generation.

$D ::= G \,\|\, D/x$	(disjunction)	$H ::= a \,\|\, e \Rightarrow F$	(hierarchy)
$G ::= g \Rightarrow a \,\|\, (G \,\|\, G)$	(clause)	$F ::= H \,\|\, (H \,\|\, F)$	(forest)

In Table IV, one easily observes the interplay between the disjunctive and hierarchical forms of a process. The process $\mathcal{D}_P$ is obtained from the initial specification P by a structural decomposition of its propositions. Using information carried by the clock relation $\preceq_P$, one iteratively picks the elementary guarded commands $g \Rightarrow a$ of $\mathcal{D}_P$ that match the successive clocks of $\preceq_P$ to form its hierarchical decomposition.

Table IV. Disjunctive and hierarchical representations of a process

(process)	(disjunctive form)	(clock relation)	(hierarchical form)
$s^0 = 0$ $\hat{x} = \hat{s}$ $s' := \neg s$ $s = \hat{y}$ $y := s$	$s^0 = 0$ $\hat{x} \Rightarrow \hat{s}$ $\hat{s} \Rightarrow \hat{x}$ $\hat{s} \Rightarrow s' := \neg s$ $s \Rightarrow \hat{y}$ $\hat{y} \Rightarrow s$ $s \Rightarrow y := s$	$\hat{x}, \hat{s}$ $/ \quad \setminus$ $\neg s \qquad s, \hat{y}$	$s^0 = 0$ $\hat{x} = \hat{s}$ $\hat{x} \Rightarrow \left(\begin{array}{l} s' := \neg s \\ \hat{y} = s \\ s \Rightarrow y := s \\ \neg s \Rightarrow 1 \end{array} \right)$

The disjunctive form of a process P is defined by application of the function $\mathcal{D}_P =^{\text{def}} \mathcal{D}[\![P]\!]^1$ in Figure 2. We observe that we have $[\![\mathcal{D}_P]\!] = [\![P]\!]$ by construction. In Appendix it is used to easily identify the input and output signals of a process, or, e.g., in Definition 12 to define conflict-free transitions. The hierarchical form of a process P is defined

by application of the function $\mathcal{H}_P \stackrel{\text{def}}{=} \mathcal{H}[\![P]\!]^1_{\preceq}$ in Figure 2. It takes an initial clock 1 and the clock relation $\preceq_P$ as parameters. This relation is iteratively decomposed using the function top $\preceq$, which returns the clock equivalence class H on top of $\preceq$, and the function bot $\preceq$, which returns the set of clock relations under H in $\preceq$.

$$\begin{array}{rlrl}
\mathcal{D}[\![l = r]\!]^g &= g \Rightarrow l = r & \mathcal{D}[\![f \Rightarrow P]\!]^e &= \mathcal{D}[\![P]\!]^{e \wedge f} \\
\mathcal{D}[\![x \rightarrow l]\!]^g &= g \wedge \hat{x} \Rightarrow x \rightarrow l & \mathcal{D}[\![P/x]\!]^e &= (\mathcal{D}[\![P]\!]^e)/x \\
\mathcal{D}[\![a]\!]^{e \vee g} &= \mathcal{D}[\![a]\!]^e \,|\, \mathcal{D}[\![a]\!]^g & \mathcal{D}[\![P \,|\, Q]\!]^e &= \mathcal{D}[\![P]\!]^e \,|\, \mathcal{D}[\![Q]\!]^e
\end{array}$$

$$\begin{array}{rl}
\mathcal{H}[\![\prod_{i=1}^n g_i \Rightarrow a_i]\!]^g_{\preceq} &= \prod_{i \in I} a_i \,|\, \mathsf{sync}^H \,|\, h \Rightarrow F \\
\text{where } h \in H &= \text{top } \preceq \\
I &= \{0 < i \leq n \,|\, \hat{P} \models g \wedge h = g_i \wedge a_i \notin H\} \\
F &= \prod_{\preceq_j \in \text{bot} \preceq} \mathcal{H}[\![\prod_{i \notin I} g_i \Rightarrow a_i]\!]^{g \wedge h}_{\preceq_j}
\end{array}$$

Figure 2. Disjunctive and hierarchic normals form $\mathcal{D}_P$ and $\mathcal{H}_P$ of a process P

The algorithm $\mathcal{H}$ iteratively constructs a subset I of the indexed clauses $g_i \Rightarrow a_i$ of the disjunctive form $\mathcal{D}_P$. It selects propositions $g_i \Rightarrow a_i$ whose guard g_i matches g, the clock on the path to the present hierarchy $\preceq$, and h, the clock of the present hierarchy $\preceq$. It discards those propositions $g_i \Rightarrow a_i$ whose action a_i is redundant with a clock expression of H. The iteration continues breadth-first to define the forest corresponding to bot $\preceq$ using the remaining propositions $g_i \Rightarrow a_i, i \notin I$. We write $\mathsf{sync}^{h_1,h_2} \stackrel{\text{def}}{=} h_1 = h_2$ to synchronize the clock of the equivalence class H of top $\preceq$. Again, one easily observe that, by construction, we have $[\![\mathcal{H}_P]\!] = [\![P]\!]$ thanks to the semantics information provided by the clock relation $\preceq$.

5. Formal analysis of processes

A hierarchical process satisfies necessary conditions for the property of *endochrony* to hold. A process is said endochronous (Definition 8) iff, given an external (asynchronous) stimulation of its inputs, it reconstructs a unique synchronous behavior (up to clock-equivalence). In other words, endochrony denotes the class of processes that are stallable or insensitive to internal and external propagation delays: patient processes, in the sense of [6].

DEFINITION 8 (endochrony [12]).
P is endochronous iff $\forall b, c \in [\![P]\!]$, $(b|_{\text{in}(P)}) \approx (c|_{\text{in}(P)}) \Rightarrow b \sim c$.

In the Polychrony workbench [16], GALS architectures are modeled starting from the property of endochrony and the criterion of *endo-isochrony* [12]: two endochronous processes P and Q of interface $I = \text{vars}(P) \cap \text{vars}(Q)$ are endo-isochronous iff $(P|_I) \,|\, (Q|_I)$ is endochronous. This criterion ensures that the refinement of the specification $P \,|\, Q$ by a distributed implementation $P \parallel Q$ is semantics-preserving. Although restrictive, it is directly amenable to static verification.

5.1. Formal analysis of hierarchical processes

Starting from the algorithm specified in Definition 6 and the class of hierarchical processes isolated in Definition 7, we seek towards processes which satisfies the property of endochrony in the context of the iSTS algebra by introducing the notion of controllability, Definition 13.

A correct definition of controllability requires a precise partition of the signals of a process P into input signals $\text{in}(P)$, output signals $\text{out}(P)$ and state variables $\text{def}(P)$. This partition is defined in Appendix. A signal $x \in \text{out}(P)$ (resp. $x \in \text{def}(P)$) is an output (resp. state) of P iff, whenever its presence is implied by a clock e, then there exists another clock f implied by e whose action is $x := y$ (resp. $x' := y$) for some reference y. The disjoint sets $\text{def}(P)$ and $\text{out}(P)$ define the input signals $\text{in}(P)$ of a process by their complementary in $\text{vars}(P)$.

Definability A controllable process needs to correctly define its local and bound signals in order to meet the property of determinism, and hence endochrony. For instance, let us reconsider (left) the example of the specification R, below.

$$R \stackrel{\text{def}}{=} (s \Rightarrow x \,|\, \neg s \Rightarrow y)/s \quad Q \stackrel{\text{def}}{=} (s^0 = 1 \,|\, s' := z \,|\, s \Rightarrow x := s \,|\, \neg s \Rightarrow y := s)/s$$

It may seem controllable by apparently having two output signals x and y defined upon the value of the signal s. However, notice that s is left undefined. Hence the behavior of R is non-deterministic, since the choice of the value of s is free. Had s been associated to some external input z, as in Q (right), one could have concluded that it is indeed controllable. In fact, notice that Q is a refinement of R: $[\![Q]\!] \subseteq [\![R]\!]$.

DEFINITION 9. *A process P is* well-defined *iff all bound variables of P are output signals or state variables. By induction on P,*

- *an atomic proposition a is well-defined.*
- *a guarded command $e \Rightarrow P$ is well defined iff P is well-defined*
- *a composition $P \,|\, Q$ is well-defined iff P and Q are well-defined*
- *P/x is well-defined iff P is well-defined and $x \in \text{def}(P) \cup \text{out}(P)$.*

Causality Causality is another possible origin of non-determinism. Let us consider the example of a clock proposition $\hat{x} =^{\text{def}} (x = x)$. It does not define an output signal, it just says that the signal x is present. It can be regarded as an abstraction of the proposition $x := y \stackrel{\text{def}}{=} (x = y \,|\, y \rightarrow x)$, which defines x as an output signal and assigns the value of the input y to it. What happens next, if we additionally write $y = x$ or $y := x$? In the first case, one can still regard the process $x := y \,|\, y = x$ as a redundant yet deterministic specification: x takes the value of y. This is not the case in the latter specification, which tries to mutually define the output signals x and y: $x := y \,|\, y := x$. It is a non-deterministic specification, as both true and false are possible values of x and y every time this is needed. Thanks to the definition of $x := y$ by $x = y \,|\, y \rightarrow x$, the analysis of causal specifications relies on the information provided by the scheduling relation of a process P: a cyclic definition of output signals yields a proposition $e \Rightarrow x \rightarrow x$ at some clock e in the transitive closure of the scheduling relation implied by $\hat{P}$. Hence Definition 10.

DEFINITION 10 (causality).
P is non-causal *iff* $(\hat{P} \models e \Rightarrow x \rightarrow x) \Rightarrow (\hat{P} \models e = 0)$.

Determinism The isolation of well-defined and non-causal processes allows to touch an important intermediate result. When a process both defines its bound variables from the value of free variables and does not define its outputs in a cyclic manner, it is deterministic in the sense of Definition 11.

DEFINITION 11 (determinism).
P is deterministic *iff* $\forall b, c \in [\![P]\!]$, $(b|_{\text{in}(P)}) = (c|_{\text{in}(P)}) \Rightarrow b = c$.

PROPERTY 5 (determinism).
If P is well-defined and non-causal then P is deterministic

One may be surprised by the absence of a notion of conflict-freedom in Definition 11. For instance, consider the process $P =^{\text{def}} e \Rightarrow x = r \,|\, f \Rightarrow x = s$, one may wonder why the invariant $Q =^{\text{def}} e \wedge s \Rightarrow s = r$ is not required to check P deterministic. However, a careful attention to the denotational semantics of P shows that $[\![P]\!] = [\![P \,|\, Q]\!]$ since synchronous composition filters conflicting behaviors. Checking this property is nonetheless mandatory to specify a correct sequential code generation scheme. In the Polychrony workbench, the resolution of conflicting specifications is performed by the synthesizing a proof obligation Q which is called a *clock constraint*:

DEFINITION 12. *P is* conflict-free *iff* $\mathcal{D}_P =^{\text{def}} (\prod_{i=1}^{m} g_i \Rightarrow a_i)\,/x_{1..n}$ *and for all* $0 < i, j \leq m$, $a_i =^{\text{def}} (l := x)$ *and* $a_j =^{\text{def}} (l := y)$ *implies* $\hat{P} \models g_i \wedge g_j \Rightarrow x = y$.

Controllability If Definition 11 is compared to the behavior depicted in Example 1, one easily notices that a deterministic process still misses some important capabilities to meet endochrony. A deterministic process must have a way to initiate a reaction upon the arrival of a pre-determined input signal, to check whether it needs further inputs and finally produce an output. The notion of hierarchy allows to fill this remaining gap. A hierarchical process P has a pre-determined input tick, the minimum $\min \preceq_P$ of its clock relation. Upon the presence of the tick, it is able to decide whether an additional input is needed or not, it is able to decide whether it should produce outputs. However, controllability requires this decision to be made starting from the input signals of a process (and not its output signals or state variables).

DEFINITION 13 (controllability). *A process P is* controllable *iff P is well-defined, non-causal, hierarchical and for all $y \in \mathrm{vars}(P)$, there exists $x \in \mathrm{in}(P)$ such that $x \preceq_P y$.*

Definition 13 bridges the remaining gap from a deterministic specification to an endochronous program. Property 6 generalizes the proposition of [12] to the context of partially defined signals of iSTS.

PROPERTY 6. *If P is controllable then P is endochronous.*

6. Correct-by-construction protocol synthesis

Building upon iSTS and its implementation in the Polychrony workbench, we seek towards a formal methodology to characterize the invariants of protocol synthesis and propose the use of global model transformation to synthesize specialized and optimized component wrappers using the behavioral type information.

6.1. A formal design methodology

To this end, we characterize a correct-by-construction methodology for the compositional design of architectures starting from elementary endochronous processes. We focus on the definition of a refinement methodology and semantics preservation criterion that naturally proceed from the model of polychrony and define the spectrum of bounded protocol synthesis in system design. Refinement-based design in the Polychrony workbench using the criterion of flow-invariance has been studied in a number of related works on system design [12, 15].

In the present study, we chose to restrict ourselves to the scope of polychrony excluding unbounded asynchrony. We therefore define the following criteria, which recast endochrony and flow-invariance to the context of finite flow-equivalence. We say that a process P is finite flow-preserving iff given finite flow-equivalent inputs, it can only produce behaviors that are finite flow equivalent.

DEFINITION 14 (finite flow-preservation).
P *is* finite flow-preserving *iff* $\forall b, c \in [\![P]\!]$, $(b|_{\mathrm{in}(P)}) \approx^* (c|_{\mathrm{in}(P)}) \Rightarrow b \approx^* c$.

Example of finite flow-preserving processes are endochronous processes. An endochronous process which receives finite flow equivalent inputs produces clock-equivalent outputs. It hence forms a restricted subclass of finite-flow preserving processes. Also note that flow-preservation is stable to the introduction of a finite buffering protocol.

PROPERTY 7 (finite flow-preservation).
1. If P is endochronous then P is finite flow preserving.
2. If P is finite flow-preserving then $\mathsf{fifo}_N\langle P\rangle$ *is finite flow-preserving.*

A refinement-based design methodology based on the property of finite flow-preservation consists of characterizing sufficient invariants for a given model transformation to preserve flows.

DEFINITION 15 (finite flow-invariance).
The transformation of P into Q such that $I \subset \mathrm{in}(P) = \mathrm{in}(Q)$ is finite flow-invariant *iff* $\forall b \in [\![P]\!]$, $\forall c \in [\![Q]\!]$, $(b|_I) \approx^* (c|_I) \Rightarrow b \approx^* c$.

The property of finite flow-invariance is a very general methodological criterion. For instance, it can be applied to the characterization of correctness criteria for model transformations such as protocol insertion or desynchronization. Let P and Q be two finite flow-preserving processes and R a protocol to link P and Q, such as a finite FIFO buffer fifo_N, or a double hand-shake protocol, or a relay station [6], or a loosely time-triggered architecture [4].

DEFINITION 16 (flow-preserving protocol). *The process R is a flow-preserving protocol iff there exists $n > 0$ such that inputs* $\mathrm{in}(R) = \{x_{1..n}\}$ *are finite flow-equivalent to outputs* $\mathrm{out}(R) = \{y_{1..n}\}$

$$\forall b \in [\![R]\!], b|_{x_{1..n}} \approx^* (b|_{y_{1..n}}[x_i/y_i]_{0<i\leq n})$$

Wrapping a process P with a protocol R, written $R\langle P\rangle$, is defined by redirecting the signals of P to R using substitutions.

DEFINITION 17 (wrapper). *Let P be a process such that* $\mathrm{in}(P) = \{x_{1..m}\}$ *and* $\mathrm{out}(P) = \{x_{m+1..n}\}$. *Let R be a flow-preserving protocol such that* $\mathrm{in}(R) = \{y_{1..n}\}$ *and* $\mathrm{out}(R) = \{z_{1..n}\}$. *The wrapper of P with R is the template process noted $R\langle P\rangle$ and defined by:*

$$R\langle P\rangle \stackrel{\text{def}}{=} \left(\begin{array}{c} ((R[x_i/z_i]_{m<i\leq n})\,[x_i/y_i]_{0<i\leq m}) \\ \mid \\ ((P[y_i/x_i]_{m<i\leq n})\,[z_i/x_i]_{0<i\leq m}) \end{array} \right) /y_{1..n}z_{1..n}$$

A sufficient condition for the insertion of a protocol between two synchronous processes P and Q to finite preserve flow is to guaranty that $P|_I \,|\, Q|_I$ is finite flow preserving for $I = \mathrm{vars}(P) \cap \mathrm{vars}(Q)$, meaning that all communications between P and Q via a shared signal $x \in I$ should be flow preserving and that P and Q may otherwise evolve independently. This condition is stated by Property 8.

PROPERTY 8 (protocol insertion). *If R is a flow-preserving protocol and P is finite flow-preserving then $R\langle P\rangle$ is finite flow-preserving. If R is a flow-preserving protocol and P, Q, $P|_I \,|\, Q|_I$ are finite flow-preserving then $R\langle P\rangle \,|\, R\langle Q\rangle$ is finite preserving ($I = \mathrm{vars}(P) \cap \mathrm{vars}(Q)$).*

6.2. Case-study of latency-insensitive protocols

To demonstrate the extent of a methodology based on flow-preservation, we consider the case-study of latency-insensitive protocols [6]. In the model of latency-insensitive protocols [6], architecture components are denoted by the notion of *pearls* ("intellectual property under a shell") and are required to satisfy an invariant of *patience* (robustness to external delays). A *latency-insensitive protocol* consists of the client-side controller of a patient or endochronous process P which aims at guarantying the preservation of flows between P and the network Q.

The construction of a latency-insensitive protocol for a process P consists of associating a channel to each signal x of the process P. A channel chan_P^x consists of a carrier, denoted by data_x, of a clock, denoted by clk_x, and of a stop signal, noted stop_x. The bus multiplexes the signal x into the carrier data_x, which repeatedly sends the current value of x, and the clock clk_x, which is true iff x is present.

The signal stop_x can be used to either inhibit the calculation of the process P (which is the server of x) or to forward the status of x (to a client Q of P) by synchronizing it to the master clock tick_P (resp. tick_Q). Since we wish to use channels to wrap both input and output signals, the definition does not provide scheduling relation but just equational propositions between the signal x and its channel's bundle.

$$\mathsf{chan}_P^x \stackrel{\text{def}}{=} \left(\begin{array}{rl} \mathsf{clk}_x \Rightarrow \mathsf{data}_x & = x \\ \mathsf{clk}_x & = \hat{x} \\ \widehat{\mathsf{tick}_P} \Rightarrow \mathsf{stop}_x & = 0 \end{array} \right) \quad s.t.\ \ \mathsf{tick}_P \in \{x \,|\, \hat{x} \in \min(\preceq_P)\} \cap \mathrm{in}(P)$$

The relay station from a channel chan^x to the channel chan^y is defined by the generic process $\mathsf{relay}_1^{x,y}$. Initially, the relay is stopped (proposition $\mathsf{stop}_x^0 = 1$) and clock of the output y is false (proposition $\mathsf{clk}_y^0 = 0$) meaning that the initial data_y^0 is irrelevant (and does not have to be specified). Then, the next value of stop_x is set to the present value of stop_y (proposition $\mathsf{stop}'_x := \mathsf{stop}_y$) meaning that the stop signal is propagated with a delay from the output channel y to the input x.

$$\mathsf{relay}_1^{x,y} \stackrel{\text{def}}{=} \left(\begin{array}{l} \left(\mathsf{clk}_y^0 = 0 \,|\, \mathsf{stop}_x^0 = 1 \,|\, \mathsf{stall}^0 = 0\ \right) \\ \left| \begin{array}{l} \mathsf{stop}'_x := \mathsf{stop}_y \\ \mathsf{stall}' := \mathsf{stop}_x \wedge \mathsf{stop}_y \\ \neg\mathsf{stall} \Rightarrow (\mathsf{data}'_y := \mathsf{data}_x \,|\, \mathsf{clk}'_y := \mathsf{clk}_x) \\ \mathsf{stall} \Rightarrow (\mathsf{data}'_y := \mathsf{data}_y \,|\, \mathsf{clk}'_y := 0) \end{array} \right) /\mathsf{stall} \end{array} \right)$$

The next state of the relay is a stall (variable stall') if its input channel requests it to stop (stop_x is true) and did already stop at the previous period (stop_y is true). If the relay is not stall (guard $\neg\mathsf{stall}$) then the next packet of the output channel y will be the present packet of input channel x (propositions $\mathsf{data}'_y := \mathsf{data}_x$ and $\mathsf{clk}'_y := \mathsf{clk}_x$). If the relay is stall (guard stall) then it transmits a *void* packet (proposition $\mathsf{data}'_y := \mathsf{data}_y$, setting the output clock clk'_y to false).

Connecting a relay station to a process P amounts to constrain every input-output signal x of P to satisfy the invariants of the channel chan^x. An input signal x of P will be present iff not delayed by the network Q (proposition $\mathsf{clk}_x = \hat{x}$) and only upon request (proposition $\mathsf{tick}_P \Rightarrow \neg\mathsf{stop}_x$). An output signal x of P will be made available to the network Q only if it is present in P (proposition $\mathsf{clk}_x = \hat{x}$) and only if it is requested by the receiver (proposition $\mathsf{tick}_P \Rightarrow \neg\mathsf{stop}_x$). Just as for a FIFO buffer, it is easy to assemble relays to form larger protocols:

$$\begin{aligned} \mathsf{relay}_n^{x,y} &\stackrel{\text{def}}{=} (\mathsf{relay}_1^{x,z} \,|\, \mathsf{relay}_{n-1}^{z,y})/z,\ \forall m, n > 1 \\ \mathsf{relay}_n^{x_{1..m},y_{1..m}} &\stackrel{\text{def}}{=} (\mathsf{relay}_n^{x_{1..m-1},y_{1..m-1}} \,|\, \mathsf{relay}_n^{x_m,y_m}) \end{aligned}$$

and define the wrapper of a process P by the template process implementing a relay station connected to as many channels as free variables in P. The wrapper $\mathsf{relay}_N\langle P\rangle$ consists of redirecting the input signals $\mathrm{in}(P) = x_{1..m}$ and output signals $\mathrm{out}(P) = x_{m+1..n}$ of P to an input relay station $\mathsf{relay}_N^{x_{1..m},y_{1..m}}$ and an output relay station $\mathsf{relay}_N^{y_{m+1..n},x_{m+1..n}}$ by substituting the signal $x_{1..n}$ with fresh ones $y_{1..n}$ in P.

$$\mathsf{relay}_N\langle P\rangle \stackrel{\text{def}}{=} \left(\left(\begin{array}{l} \mathsf{relay}_N^{x_{1..m},y_{1..m}} \\ \left| \ \mathsf{chan}_P^{y_{1..m}} \right. \end{array} \right) \,|\, (P[y_i/x_i]_{i=1}^n) \,|\, \left(\begin{array}{l} \mathsf{chan}_P^{y_{m+1..n}} \\ \left| \ \mathsf{relay}_N^{y_{m+1..n},x_{m+1..n}} \right. \end{array} \right) \right) /y_{1..n}$$

From [6], we easily observe that the relay satisfies the expected correctness criterion of finite flow-preservation, by inhibiting calculations

on either side of the network upon the unavailability of a required signal. Since our model encompasses scheduling specifications, Property 9 additionally requires $P \,|\, Q$ be non-causal for flow-preservation to hold.

PROPERTY 9. *If P is finite flow-preserving then $\mathsf{relay}_N\langle P\rangle$ is finite flow-preserving. If P, Q are finite flow-preserving and $P \,|\, Q$ is non-causal then $\mathsf{relay}_N\langle P\rangle \,|\, \mathsf{relay}_N\langle Q\rangle$ is finite flow-preserving.*

An advantage of the methodology defined by Property 9 compared to that defined by Property 8 on protocol insertion is that the requirement on $P \,|\, Q$ is limited, and hence that correctness can easily be satisfied by construction. A potential draw-back of the approach is that a process P will stall, by having its master clock inhibited by a stop^x signal, as soon as one of its clients Q will stall, or will not be able to transmit x fast enough. To prevent a possible cascade of stalls, a careful dimensioning of buffers is hence necessary to meet timing requirements.

Our approach departs from a latency-insensitive protocol by partly revealing the structure of the *pearl* (the IPs) using P as a behavioral type and by giving a multi-clocked specification to the *shell* (the wrapper). We may consider a variant $\mathsf{bus}_N\langle P\rangle$ of the protocol where the signal stop^x only inhibits the source signal x. The absence of x can be propagated back to the server P and then be hierarchically propagated in $\mathcal{H}_P$, allowing it to perform another action independently of x, e.g., interact with another non-stalled client. This can easily be done by considering the following multiplexer:

$$\mathsf{bus}^x \stackrel{\text{def}}{=} (\mathsf{clk}_x \Rightarrow \mathsf{data}_x = x \,|\, \mathsf{clk}_x = \hat{x} \,|\, \hat{x} \Rightarrow \mathsf{stop}_x = 0)$$

The matching flow-preserving protocol is defined by using relays to buffer and transmit data. Notice that, in $\mathsf{relay}\langle P\rangle$, one should synchronize the clock of all stop^x signals to define a minimum in the hierarchy of $\mathsf{relay}\langle P\rangle$, whereas in $\mathsf{bus}_N\langle P\rangle$, the clock of every stop^x appears above $\hat{x}$ and helps to control the communication of the process P.

$$\mathsf{bus}_N\langle P\rangle \stackrel{\text{def}}{=} \left(\left(\begin{array}{l}\mathsf{relay}_N^{x_{1..m},y_{1..m}}\\ \mid\ \mathsf{bus}^{y_{1..m}}\end{array}\right) \mid (P[y_i/x_i]_{i=1}^n) \mid \left(\begin{array}{l}\mathsf{bus}^{y_{m+1..n}}\\ \mid\ \mathsf{relay}_N^{y_{m+1..n},x_{m+1..n}}\end{array}\right)\right)/y_{1..n}$$

We meet the following instance of Property 8, which requires the interaction between P and Q to be flow-preserving, but results in a reduced latency. By contrast, notice that $\mathsf{relay}\langle P\rangle|_I \,|\, \mathsf{relay}\langle Q\rangle|_I$ satisfies flow-preservation by design, because it inhibits the master clock of both P and Q upon delayed transmission along the signals of I.

PROPERTY 10. *If P is finite flow-preserving then $\mathsf{bus}_N\langle P\rangle$ is finite flow-preserving. If P, Q are finite flow-preserving and $P|_I \,|\, Q|_I$ is finite flow-preserving (for $I = \mathrm{vars}(P) \cap \mathrm{vars}(Q)$) then $\mathsf{bus}_N\langle P\rangle \,|\, \mathsf{bus}_N\langle Q\rangle$ is finite flow-preserving.*

6.3. A perspective in controller protocol synthesis

Placed in the context of protocol synthesis, where the specification of a *pearl* or IP of behavioral type P may not necessarily be available, the compilation of the behavioral type P with its wrapper (relay_N or bus_N) provides a *controller* of the pearl, that satisfies the expected invariant of patience yet in a way that is cautious of the internal behavior of the IP, as expressed by its behavioral type P.

This results in the synthesis of an optimized protocol, that consists of the hierarchization of the behavioral type P and of its wrapper $\mathsf{relay}_N\langle P\rangle$ or $\mathsf{bus}_N\langle P\rangle$, allowing for the correct-by-construction plug of the pearl of type P in the architecture.

In the methodology of latency-insensitive design, the pearl considered for composition in a network of characteristics Q is required to satisfy the invariant of being patient or stallable. In the polychronous model of computation, this invariant can be defined by the most abstract process satisfying the criterion of controllability. It is a process of master clock tick_P and input-output signals $x_{1..n}$ defined by:

$$\mathsf{patient}^{x_{1..m},y_{1..n}} \stackrel{\text{def}}{=} \left(\neg\mathsf{stop}^x \Rightarrow \widehat{\mathsf{tick}}_P \,|\, \widehat{\mathsf{tick}}_P = \vee_{i=1}^n \hat{x}_i\right)$$

An endochronous process P naturally satisfies the denotational containment relation $[\![P]\!] \subseteq [\![\mathsf{patient}^{\mathrm{vars}(P)}]\!]$. It defines a controller that has the capability of stopping output relays and of being interrupted by input relays in a way that is contextually related to the state and control of the pearl it is the type of. This naturally result in a reduced latency. By contrast, a generic latency-insensitive protocol requires all data to be retransmitted and resynchronized at each clock cycle or conservatively stall.

7. Related works

Behavioral abstraction Our methodology of scalable and correct-by-construction exploration of abstraction/refinement of system behaviors is shared with the work of Henzinger et al. on interface automata [8]. Our approach primarily differs from interface automata in the data-structure considered: clock equations, boolean propositions and state variable transitions express the multi-clocked synchronous behavior of a system. Compared to an automata-based approach, our declarative approach allows to hierarchically explore abstraction capabilities and to cover design exploration with the methodological notion of refinement along the whole design cycle of the system, ranging from the early

requirements specification to the latest sequential and distributed code-generation [15, 12]. Our contribution contrasts from related studies by the capability to capture scalable abstractions of the system. In our type system, scalability ranges from the capability to express the exact meaning of the program, in order to make structural transformations and optimizations on it, down to properties expressed by boolean equations between clocks, allowing for a rapid static-checking of design correctness properties. Our system further allows for correct-by-construction design abstraction and refinement patterns to be applied on a model, e.g. abstraction of states by clocks, abstraction of existentially quantified clocks, hierarchical abstraction, in the aim of choosing an optimal degree of abstraction for a faster verification.

Synchrony and asynchrony Synchrony is a popular computational model in hardware design. Desynchronization converts that model into a more general one, a GALS model, that is suitable for system-on-chip design. Therefore, one may naturally consider investigating the links between these two models further, understand them as Ptolemy domains [5], and study the refinement-based design of GALS architectures starting from polychronous specifications captured from heterogeneous elementary components. The aim of capturing both synchrony and asynchrony in a unifying model of computation is shared by several approaches: communicating sequential processes [9], Kahn networks [10], latency insensitive protocols [6], heterogeneous systems [3]. These models partition systems into synchronous islands (the *pearls*) and asynchronous networks. Synchrony and asynchrony are not partitioned in the present model. Both are captured within the same partially ordered trace structure, and related by the clock and finite-flow equivalence relations $\sim$ and $\approx^*$. The iSTS algebra carry over this unified model to capture modeling, transformation and verification of embedded systems from the highest levels of requirements specification down to its clock-accurate implementation as a GALS architecture.

8. Conclusions

In the theory of latency-insensitive protocols [6], the status of an architecture components is rendered by the notion of *pearls* ("intellectual property under a shell") and assumed to satisfy the invariant of *patience* (robustness to external delays) by interfacing it to a generic wrapper. Our approach consists of compiling a behavioral model of the IP with a wrapper that controls it. This approach still satisfies the expected invariant of patience yet requires to capture the internal behavior of the IP. This results in the synthesis of an optimized protocol allowing

for a correct-by-construction architecture construction with minimized local latency and maximized global throughput.

In conclusion, the latency-insensitive protocol approach can be seen as a black-box approach, where a conservative yet generic wrapper is connected to a given IP to ensure its functional correctness in the architecture, whereas the desynchronization based approach is a grey-box approach where part of the IP's behavior need to be specified to synthesize an architecture sensitive wrapper with its environment and ensure functional correctness. Our presentation is based on a high-level modeling and specification methodology that ensures compositional correctness through an algebra capturing behavioral aspects of component interfaces. This minimalist algebra, called iSTS, specifies the state transitions, synchronization relations and scheduling constraints implied by a given system, in the presence of multiple clocks.

References

1. Amagbegnon, T. P., Besnard, L., Le Guernic, P. "Implementation of the data-flow synchronous language Signal". In *Conference on Programming Language Design and Implementation*. ACM Press, 1995.
2. Hoe, J., Arvind. "Synthesis of Operation-Centric Hardware Descriptions". *Proceedings of International Conference on Computer Aided Design*. IEEE Press, November 2000.
3. Benveniste, A., Caspi, P., Carloni, L. P., Sangiovanni-Vincentelli, A. L. "Heterogeneous Reactive Systems Modeling and Correct-by-Construction Deployment". In *Embedded Software Conference*. Lecture Notes in Computer Science, Springer Verlag, October 2003.
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*. Lecture Notes in Computer Science, Springer Verlag, October 2002.
5. Buck , J.T., Ha, S. , Lee, E., and Messerschmitt, D. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. In *International Journal of Computer Simulation, special issue on "Simulation Software Development"*v. 4, pp. 155-182. Ablex, April 1994.
6. Carloni, L. P., McMillan, K. L., Sangiovanni-Vincentelli, A. L. "Latency-Insensitive Protocols". In *Proceedings of the 11th. International Conference on Computer-Aided Verification*. Lecture notes in computer science v. 1633. Springer Verlag, July 1999.
7. Dijkstra, E. "A Discipline of Programming". Prentice Hall, 1976.
8. De Alfaro, L., Henzinger, T. A. "Interface theories for component-based design". *International Workshop on Embedded Software*. Lecture Notes in Computer Science v. 2211. Springer-Verlag, 2001.
9. Hoare, C. *Communicating sequential processes*. Prentice Hall, 1985.
10. Kahn, G. The semantics of a simple language for parallel programming. In Ifip *Congress*. North Holland, 1974.

11. Lee, E., 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.
12. Le Guernic, P., Talpin, J.-P., Le Lann, J.-L. Polychrony for system design. In *Journal of Circuits, Systems and Computers. Special Issue on Application-Specific Hardware Design*. World Scientific, 2002.
13. Nowak, D., Talpin, J.-P., Le Guernic, P. "Synchronous structures". In *International Conference on Concurrency Theory*. Lecture Notes in Computer Science, Springer Verlag, August 1999.
14. Pnueli, A., Shankar, N., Singerman, E. Fair synchronous transition systems and their liveness proofs. *International School and Symposium on Formal Techniques in Real-time and Fault-tolerant Systems*. Lecture Notes in Computer Science v. 1468. Springer Verlag, 1998.
15. Talpin, J.-P., Le Guernic, P., Shukla, S., Gupta, R., and Doucet, F. "Polychrony for formal refinement-checking in a system-level design methodology". *Application of Concurrency to System Design*. IEEE Press, 2003.
16. Polychrony: `http://www.irisa.fr/espresso/Polychrony`, 2004.

Appendix

We refer to the *free variables* $\mathrm{vars}(P)$ of a process P as the set of signal names that occur free in the lexical scope of P.

$$\begin{array}{r}\mathrm{vars}(x^0 = v) = \mathrm{vars}(x = v) = \mathrm{vars}(x' = v) = \{x\}\\ \mathrm{vars}(x = y) = \mathrm{vars}(x' = y) = \mathrm{vars}(x \to y) = vars(x \to y') = \{x, y\}\\ \mathrm{vars}(e \wedge f) = \mathrm{vars}(e \vee f) = \mathrm{vars}(e \setminus f) = \mathrm{vars}(e) \cup \mathrm{vars}(f)\\ \mathrm{vars}(e \Rightarrow P) = \mathrm{vars}(e) \cup \mathrm{vars}(P) \quad \mathrm{vars}(P \,|\, Q) = \mathrm{vars}(P) \cup \mathrm{vars}(Q)\\ \mathrm{vars}(P/x) = \mathrm{vars}(P) \setminus \{x\} \quad \mathrm{vars}(0) = \mathrm{vars}(1) = \emptyset\end{array}$$

The defined state and output signals $\mathrm{def}(P)$ and $\mathrm{out}(P)$ of a process P are defined by the relation of Figure **??** starting from the disjunctive form of P (Figure 2). A signal $x \in \mathrm{out}(P)$ (resp. $x \in \mathrm{def}(P)$) is an output of P iff, whenever its presence is implied by a guard g, then there exists a guard h implied by g whose action is $x := r$ (resp. $x' := r$) for some r. The sets $\mathrm{def}(P)$ and $\mathrm{out}(P)$ define the variables $x \in \mathrm{vars}(P)$ that are the input signals $\mathrm{in}(P)$ of P. In the definition of the generic function $\mathrm{locs}(P)$, we overload $\hat{x}$ to $\hat{l}$ and assume that $\hat{x'} \stackrel{\mathrm{def}}{=} \hat{x}$.

$$\begin{array}{r}x \in \mathrm{def}(P) \Leftrightarrow x' \in \mathrm{locs}(\mathcal{D}_P) \wedge \exists 0 < i \leq n, g_i \stackrel{\mathrm{def}}{=} 1 \wedge a_i \stackrel{\mathrm{def}}{=} x^0 = v\\ x \in \mathrm{out}(P) \Leftrightarrow x \in \mathrm{locs}(\mathcal{D}_P) \wedge x \notin \mathrm{def}(P)\\ \mathrm{in}(P) \;=\; (\mathrm{vars}(P) \setminus \mathrm{def}(P)) \setminus \mathrm{out}(P)\end{array}$$

$$\begin{array}{l}l \in \mathrm{locs}\left(\left(\prod_{i=1}^{n} g_i \Rightarrow a_i\right)/x_{1..m}\right) \Leftrightarrow\\ \quad \forall 0 < i \leq n \, s.t. \, \hat{P} \models g_i \Rightarrow \hat{l}, \exists \, 0 < j \leq n, \hat{P} \models g_i \Rightarrow g_j \wedge a_j \stackrel{\mathrm{def}}{=} l := r\end{array}$$