A Synchronous Semantics of Higher-Order Processes for Modeling Reconfigurable Reactive Systems

Jean-Pierre Talpin and David Nowak

INRIA-Rennes and IRISA, Campus de Beaulieu, F-35042 Rennes

Abstract. Synchronous languages are well suited for the design of dependable real-time systems: they enable a very high-level specification and an extremely modular implementation of complex systems by structurally decomposing them into elementary synchronous processes. To maximize the support for the generic implementation and the dynamic reconfigurability of reactive systems, we introduce a semantics of higher-order processes in a seamless extension of the synchronous language SIGNAL. To enable the correct specification and the modular implementation of systems, we introduce an expressive type inference system for characterizing their temporal and causal invariants.

1 Introduction

Synchronous languages, such as SIGNAL [2], LUSTRE [9], ESTEREL [3] have been specifically designed to ease the development of reactive systems. The synchronous hypothesis provides a deterministic notion of concurrency where operations and communications are instantaneous. In a synchronous language, concurrency is meant as a logical way to decompose the description of a system into a set of elementary communicating processes. Interaction between concurrent components is conceptually performed by broadcasting events. Synchronous languages enable a very high-level specification and an extremely modular design of complex reactive systems by structurally decomposing them into elementary processes. The compilation of such specifications consists of the proof of global safety requirements and of the assembly of processes to form automata or circuits of fixed interaction topology. In practice, however, fixing the topology of a system makes it difficult to model practical situations requiring an optimal reaction to a major environmental change, e.g., the tele-maintainance of processing algorithms on a satellite; the adaptation of an event handler to a context-sensitive, menu-driven, user-interface; loading an applet in a browser; the dynamic configuration of a flight-assistance system with respect to the flight phase. Reconfigurations should be taken into account early in the design of the system in order to prevent accidental alteration of services or interferences during execution. The reactivity and safety invariants of the system should be preserved. A simple and intuitive way to express the dynamic configurability and the generic assembly of reactive system components in a synchronous language is to introduce a semantics of higher-order processes. Higher-order processes have been extensively studied in the context of process calculi [8, 14] and allow reasoning on state

and mobility in distributed system. Contemporary work has demonstrated the practicality of higher-order programming languages for hardware-design (e.g. LAVA [4], HAWK [13]) and the introduction of LUCID-synchrone [6] has been a first step towards the generalization of the principles of synchronous programming to higher-order languages. To model reconfigurable reactive systems, we give a seamless extension of the synchronous language SIGNAL with first-class processes (section 2). We introduce an inference system (section 3 and 4) for determining the temporal and causal invariants of processes. We state the formal properties of our inference systems (section 5).

2 Semantics

We adopt the synchronous programming model of SIGNAL [2] and regard a reactive system as the composition of co-iterative simultaneous equations on discrete signals. A signal represents an infinite sequence of values defined over a discrete and totally ordered set of instants. Each equation in the system specifies an elementary and instantaneous operation on input signals which produces outputs. An instant is the time elapsed between two successive occurrences of messages at the signal. This simple and intuitive notion of time allows concurrency to meets determinism: *synchronous composition* concerns signals present at the same time. The systematization of this hypothesis has several immediate consequences: communication becomes broadcast and is immediate (inputs and outputs are present at the same time). But it also imposes a programming style which slightly differ from conventions, in that consideration on the logical notion of time imposed by the model are essential. This can be illustrated by considering the definition of a counter from m to n: let x = (pre x m) + 1 in when (x ≤ n) x. At each instant, it is either silent or reacts to input events and provide values to its environment. The value of x is initially $x_0 = m$. At each instant $i > 0$, it is the previous value x_{i-1} of x added to 1 and is returned until the bound n is reached. In other words, $x_0 = m$ and $x_i = x_{i-1} + 1, \forall i \in [1..n - m]$.

Syntax A signal is named x, y or z. A value v is either a boolean b or a closure c. A closure c consists of a closed (i.e. s.t. $\mathrm{fv}(e) = \{x\}$) and finite expression e parameterized by a signal x. Expressions e consist of values v, pervasives p, references to signals x, abstractions proc $x\,e$, applications $e(e')$, synchronous compositions $e \mid e'$ and scoped definitions let $x = e$ in e'

x	signal	$e ::= v$	value	$p ::=$ sync	synchro
		$\mid x \mid p$	reference	$\mid$ pre	delay
$b ::=$ true $\mid$ false	boolean	$\mid e(e')$	application	$\mid$ when	sample
$c ::=$ proc $x\,e$	closure	$\mid e \mid e'$	composition	$\mid$ default	merge
$v ::= b \mid c$	value	$\mid$ let $x = e$ in e'	definition	$\mid$ reset	reset

Fig. 1. Syntax of expressions e

By extension, $x_1^{1..m_1} = e_1 \mid \ldots x_n^{m_n} = e_n$ simultaneously defines $(x_i^j)_{i=1..n}^{j=1..m_i}$ by the expressions $e_{1..n}$ of $(m_i)_{i=1..n}$ outputs. The pervasive sync $e\,e'$ synchronizes e

and e'; pre e e' references the previous value of e (initially e'); when e e' outputs e' when e is present and true; default e e' outputs the value of e when it is present or else that of e'. We write $x_{1..n}$ or $\tilde{x}$ for a sequence. We write $\mathrm{fv}(e)$ for the signals lexically free in e. For a relation R, $\mathrm{dom}(R)$ is the domain of R and $R \uplus (x, P)$ the addition of (x, P) s.t. $x \notin \mathrm{dom}(R)$ to R.

Operational Semantics We define a small step operational semantics which describes how a system e evolves over time. A step defines an instant. The evolution of the system over time is modeled by co-iteration. The relation consists of an environment E, an initial state represented by an expression e, a (composition of) result(s) $\tilde{r}$ and a final state e'. It means that e reacts to E by producing $\tilde{r}$ and becoming e'. Environments E associate signals x to results r (either absent, i.e. abs, or present, i.e. v). The axioms (val) defines the transition for values v (either v or absent to preserve stuttering), the axiom (sig) that for signals x (the result r associated to x in the environment is emitted). The rule (let) defines the meaning of let $x = e$ in e'. Since inputs and outputs are simultaneous, the hypothesis (x, r) is required to conclude that e_1 has result r (e.g., example 1). The rule (par) synchronizes the events E used to produce the result $(\tilde{r}_1, \tilde{r}_2)$ of a composition $e_1 \mid e_2$. The rules (pre$_1$) and (pre$_2$) define the meaning of a delay statement pre e_1 e_2. When the result v_1 of e_1 is present, it is stored in place of e_2 of which the result v_2 is emitted. The rules (abs$_1$) and (abs$_2$) mean that the emission of a closure c from an abstraction proc x e require all signals y free in e to be present in E.

$$E \vdash e \xrightarrow{\tilde{r}} e' \quad \text{instant} \qquad r ::= \mathsf{abs} \mid v \quad \text{result}$$
$$(E_i \vdash e_i \xrightarrow{\tilde{r}_i} e_{i+1})_{i \geq 0} \quad \text{execution} \qquad E \ni (x, r) \quad \text{environment}$$

Fig. 2. Semantics domains

$$E \vdash v \xrightarrow{v} v \quad E \vdash v \xrightarrow{\mathsf{abs}} v \quad (\mathrm{val}) \qquad \frac{E \uplus (x, r_1) \vdash e_1 \xrightarrow{r_1} e_1' \quad E \uplus (x, r_1) \vdash e_2 \xrightarrow{\tilde{r}} e_2'}{E \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \xrightarrow{\tilde{r}} \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2'} \quad (\mathrm{let})$$
$$E \uplus (x, r) \vdash x \xrightarrow{r} x \quad (\mathrm{sig})$$

$$\frac{E \vdash e_1 \xrightarrow{\tilde{r}_1} e_1' \quad E \vdash e_2 \xrightarrow{\tilde{r}_2} e_2'}{E \vdash e_1 \mid e_2 \xrightarrow{(\tilde{r}_1, \tilde{r}_2)} e_1' \mid e_2'} \quad (\mathrm{par}) \qquad \frac{E \vdash e_1 \xrightarrow{r_1} e_1' \quad E \vdash r_1(e_2) \xrightarrow{\tilde{r}} r_1'(e_2')}{E \vdash e_1(e_2) \xrightarrow{\tilde{r}} e_1'(e_2')} \quad (\mathrm{app})$$

$$\frac{E \vdash e_1 \xrightarrow{v_1} e_1' \quad E \vdash e_2 \xrightarrow{v_2} e_2'}{E \vdash \mathsf{pre}\ e_1\ e_2 \xrightarrow{v_2} \mathsf{pre}\ e_1'\ v_1} \quad (\mathrm{pre}_1) \qquad \frac{E \vdash e_2 \xrightarrow{r} e_2' \quad (x, r) \vdash e_1 \xrightarrow{\tilde{r}} e_1'}{E \vdash (\mathsf{proc}\ x\ e_1)(e_2) \xrightarrow{\tilde{r}} (\mathsf{proc}\ x\ e_1')(e_2')} \quad (\mathrm{cls}_1)$$

$$\frac{E \vdash e_1 \xrightarrow{\mathsf{abs}} e_1' \quad E \vdash e_2 \xrightarrow{\mathsf{abs}} e_2'}{E \vdash \mathsf{pre}\ e_1\ e_2 \xrightarrow{\mathsf{abs}} \mathsf{pre}\ e_1'\ e_2'} \quad (\mathrm{pre}_2) \qquad \frac{E \vdash e_2 \xrightarrow{r} e_2'}{E \vdash \mathsf{abs}(e_2) \xrightarrow{\mathsf{abs}} \mathsf{abs}(e_2')} \quad (\mathrm{cls}_2)$$

$$\frac{c = (\mathsf{proc}\ x\ e)[v/y]_{(y,v) \in E}^{y \in \mathrm{fv}(e) \backslash \{x\}}}{E \vdash \mathsf{proc}\ x\ e \xrightarrow{c} \mathsf{proc}\ x\ e} \quad (\mathrm{abs}_1) \qquad \frac{E \vdash e_1 \xrightarrow{\mathsf{abs}} e_1' \quad E \vdash e_2 \xrightarrow{r} e_2' \quad E \vdash r(e_3) \xrightarrow{\tilde{r}} r'(e_3')}{E \vdash (\mathsf{reset}\ e_1\ e_2)(e_3) \xrightarrow{\tilde{r}} (\mathsf{reset}\ e_1'\ r')(e_3')} \quad (\mathrm{rst}_1)$$

$$\frac{y \in \mathrm{fv}(e) \backslash \{x\} \quad (y, \mathsf{abs}) \in E}{E \vdash \mathsf{proc}\ x\ e \xrightarrow{\mathsf{abs}} \mathsf{proc}\ x\ e} \quad (\mathrm{abs}_2) \qquad \frac{E \vdash e_1 \xrightarrow{c} e_1' \quad E \vdash c(e_3) \xrightarrow{\tilde{r}} c'(e_3')}{E \vdash (\mathsf{reset}\ e_1\ e_2)(e_3) \xrightarrow{\tilde{r}} (\mathsf{reset}\ e_1'\ c')(e_3')} \quad (\mathrm{rst}_2)$$

Fig. 3. Operational semantics $E \vdash e \xrightarrow{\tilde{r}} e'$

The inductive definition of values v further excludes co-inductive closures, e.g. $c = \text{proc } x\, c(x)$ from $\text{let } f = \text{proc } x\, f(x)$ (the causal analysis, section 4, detects and rejects "instantaneously recursive" closures). The rule (app) evaluates e_1 to a state e_1' and result r_1. If it is absent, rule (cls_2), so is the result. If it is a closure $\text{proc } x\, e_1$, rule (cls_1), x is bound to r_2 and e_1 evaluated to e_1' and to $\tilde{r}$. The keyword reset allows to store the internal state of a process when necessary (e.g., the watch, below). If the reset e_1 is absent, rule (rst_1), no reconfiguration takes place. The initial process e_2 is evaluated to r and applied to e_3. The application produces a result $\tilde{r}$ and a new state, consisting of r' and e_3'. It is stored in place of e_2 and e_3. If e_1 emits a new process c, rule (rst_2), the previous state is discarded and the new state c' results from applying c to e_3. Pervasives sync, when and default incur no state transition. The operator default $e\ e'$ outputs the value of e when e is present. If e returns abs, the result is that of e'. The operator when $e\ e'$ outputs the value of e' when e is present with the value true. All operators on booleans (e.g. and, or, not) and on integers (e.g. +, -, *) are synchronous: inputs and outputs are present simultaneously or absent.

x	abs	v
y	abs	v'
sync $x\ y$	abs	unit

x	abs	v	v	abs
y	abs	v'	abs	v
default $x\ y$	abs	v	v	v

x	abs	v	abs	false	true
y	abs	abs	v	v	v
when $x\ y$	abs	abs	abs	abs	v

Fig. 4. Pervasives sync, when and default

Example 1. Let us consider a variant of the example 1 which reacts to an input tick by counting from an initial value start down to 0 to emit a timeout. At each instant, all expressions are simultaneously executed. The signal count is reset to start when the timeout occurs. Otherwise it takes a decrement of its previous value last. Using the timer, we define a watch that emits seconds and minutes by constructing the higher-order processes count_seconds and count_minutes). We resource to reconfiguration to store the internal state of each process (i.e. instances of count and start). Given a predefined tick signal that represents milliseconds, we assemble the watch.

```
let timer = proc (start)
     proc (tick) let count = (start when (last ≤ 1)) default (last - 1)
                  |   last = pre count start | sync (count, tick)
                  in when (last ≤ 1) true
let watch = proc (start)
     proc (tick) let count_seconds = timer(1000) | count_minutes = timer(60)
                  |   seconds = (reset (when start count_seconds) count_seconds)(tick)
                  |   minutes = (reset (when start count_minutes) count_minutes)(seconds)
                  in  seconds | minutes
```

Fig. 5. The timer and the watch

3 Temporal Invariants

As outlined in the operational semantics, the execution of processes requires the resolution of temporal relations between signals in order to ensure the respect

of synchronization constraints. Determining clock relations is usually specified in terms of a *clock calculus* [2]. A clock calculus abstracts a process by a system of boolean equations which denotes the temporal relations between signals. This system must be satisfiable in order to accept the process as an executable specification. In the spirit of an effect system [18], we express both temporal and causal invariants of synchronous processes within an expressive type system.

Type System A type τ is either a data-type bool, a type variable α, a composition $\tau \times \tau'$, a process type $\tau \to \tau'$ (of input τ and output τ'). The type of a signal x is represented by a data-type τ annotated with a clock κ. The data-type τ denotes the structure of values transported by the signal and the clock κ denotes the discrete set of instants at which data are present. A clock κ is represented by a boolean polynomial which denotes a discrete set of instants. A clock is either 0 (which denotes absence), 1 (which denotes over-sampling), a clock variable δ (which denotes a set of instants), the additive $\kappa + \kappa'$ (equivalent to the proposition $(\kappa \wedge \neg\kappa') \vee (\kappa' \wedge \neg\kappa)$), which denotes the disjoint union of the instants denoted by κ and κ', or the multiplicative $\kappa \times \kappa'$ (equivalent to $\kappa \wedge \kappa'$), which denotes the intersection of the instants denoted by κ and κ'.

$$
\begin{array}{lll}
\tau ::= \mathsf{bool} \mid \alpha \mid \tau \to \tau' \mid \tau \times \tau' \mid \tau_\kappa & \text{type} \\
\kappa ::= 0 \mid 1 \mid \delta \mid \kappa + \kappa' \mid \kappa \times \kappa' & \text{clock} \\
\sigma ::= \tau \mid \forall\alpha.\sigma \mid \forall\delta.\sigma \mid \exists!\delta.\sigma & \text{type scheme} \\
\Gamma \ni (x{:}\sigma_\kappa), \ \Delta \ni \delta & \text{hypothesis}
\end{array}
$$

$$
\overline{\Gamma}(\Delta, \tau_\kappa) = \exists\Delta\backslash\Delta'.[\forall(\mathrm{ftv}(\tau)\backslash\mathrm{ftv}(\Gamma)).\forall(\Delta'\backslash\Delta).\exists!(\Delta\cap\Delta').\tau_\kappa]s.t.\Delta' = \mathrm{fcv}(\tau)\backslash\mathrm{fcv}(\Gamma)\backslash\mathrm{fcv}(\kappa)
$$

Fig. 6. Type system

We refer to the type hypothesis Γ as a relation between signals names and polymorphic types. We refer to the control hypothesis Δ as a set of abstract clocks introduced for sampling signals (e.g., example 3). A polymorphic type σ denotes the family of all types of a signal. It is universally quantified over free type variables (written $\forall\alpha.\sigma$) and clock variables (written $\forall\delta.\sigma$) and existentially quantified over its free abstract clock variables (written $\exists!\delta.\sigma$). The generalization $\overline{\Gamma}(\Delta, \tau_\kappa)$ of the type τ_κ of a signal with respect to the type hypothesis Γ and the control hypothesis Δ is a pair $\exists\Delta''.\sigma_\kappa$ consisting of the polymorphic type of the signal and of the context-sensitive control hypothesis (i.e. which refer to signals defined in Γ). A polymorphic type σ_κ is constructed by universally quantifying τ_κ with the set of its *context-free* type variable $\alpha_{1..m}$ (i.e. free in τ but not in Γ) and clock variables $\Delta'\backslash\Delta$ (i.e. free in τ, but neither in Γ nor in κ), and by existentially quantifying it with its set of context-free control hypothesis $\Delta \cap \Delta'$ (i.e. free both in τ and Δ, but neither in Γ or κ). *Context-sensitive* control-hypothesis $\Delta'' = \Delta \backslash \Delta'$ remains. The relation of instantiation, written $\exists\Delta.\tau_\kappa \preceq \sigma_\kappa$ means that τ belongs to the family σ given the control hypothesis Δ. A pair $\exists\Delta''.\tau_\kappa$ is an instance of the polymorphic type $\forall\alpha_{1..m}\forall\Delta\exists!\Delta'.\tau'_\kappa$ iff τ equals τ' modulo a substitution from $\alpha_{1..m}$ to $\tau_{1..m}$, a substitution from Δ to $\kappa_{1..n}$ and a bijection from Δ' to Δ''.

Inference system The sequent $\Delta, \Gamma \vdash e : \tau$ inductively defines the type and clock of expressions e given hypothesis Γ and Δ. The rule (let) typechecks the definition e of a signal x with the type hypothesis Γ by giving it a type τ_κ and control hypothesis Δ. Then, the type of x is generalized as $\exists \Delta'.\sigma_\kappa$ with respect to Γ. The expression e' is then typechecked with $\Gamma \uplus (x, \sigma_\kappa)$ to built the type τ' and the control hypothesis Δ'' of e'. The conclusion accepts let e in e' as well-typed with $\Delta' \uplus \Delta''$. In the rule (var), the control hypothesis Δ introduced by instantiating the polymorphic type $\Gamma(x)$ of a signal x to τ are added to the sequent. The rule (par) for the composition $e \mid e'$ accepts two well-typed expressions e and e' and merge the disjoint control hypothesis Δ and Δ' introduced by e and e'. The rule (abs) introduces a type hypothesis τ' for x in order to accepts e as well-typed with Δ and τ. Since a process is a value, it can be assigned any clock κ provided that all signals y free in e are present (i.e. at clock κ'). The rule (app) checks the application of a process e to a signal e'. The type τ' of e' must match that expected as formal parameter by the result of e. However, no output can be present if the process itself is absent. Therefore, the clock of the output τ must be combined with κ: the presence of an output requires the presence of e (at the clock κ). This requirement is written $\tau \times \kappa$ and defined by $(\tau_\kappa) \times \kappa' = \tau_{\kappa \times \kappa'}$ and $(\tau_{1..n}) \times \kappa = (\tau_1 \times \kappa) \times \ldots (\tau_n \times \kappa)$. The types of sync and pre mean that input and output are synchronous at clock κ. The types of reset and default mean that the output is present at the clock $\kappa \vee \kappa'$ of both inputs. The clock of when defines an existentially quantified clock δ which abstracts the instants at which its first argument is present with the value true. The result is present when that clock intersects with that of the second argument: $\delta \times \kappa'$.

$$\frac{\Delta, \Gamma \vdash e : (\tau' \to \tau)_\kappa \quad \Delta', \Gamma \vdash e' : \tau'}{\Delta \uplus \Delta', \Gamma \vdash e(e') : \tau \times \kappa} \ \text{(app)} \qquad \frac{\Delta, \Gamma \vdash e : \tau \quad \Delta', \Gamma \vdash e' : \tau'}{\Delta \uplus \Delta', \Gamma \vdash e \mid e' : \tau \times \tau'} \ \text{(par)}$$

$$\frac{\exists \Delta. \tau \preceq \Gamma(x)}{\Delta, \Gamma \vdash x : \tau} \ \text{(var)} \qquad \frac{\Delta, \Gamma \vdash e : \tau \quad \Delta, \Gamma \vdash e' : \tau' \quad \Delta'', \Gamma \vdash p : \tau \to \tau' \to \tau''}{\Delta \uplus \Delta' \uplus \Delta'', \Gamma \vdash p \ e \ e' : \tau''} \ \text{(pvs)}$$

$$\frac{\Delta, \Gamma \uplus (x, \tau') \vdash e : \tau}{\Delta, \Gamma \vdash \text{proc } x \ e : (\tau' \to \tau)_{\kappa \times \kappa'}} \ \text{(abs)} \quad s.t. \ \kappa' = \Pi^{y \in fv(e) \backslash \{x\}}_{(y, \sigma_{\kappa''}) \in \Gamma} \kappa''$$

$$\frac{\Delta, \Gamma \uplus (x, \tau_\kappa) \vdash e : \tau_\kappa \quad \Delta'', \Gamma \uplus (x, \sigma_\kappa) \vdash e' : \tau'}{\Delta' \uplus \Delta'', \Gamma \vdash \text{let } x = e \text{ in } e' : \tau'} \ \text{(let)} \quad s.t. \ \overline{\Gamma}(\Delta, \tau_\kappa) = \exists \Delta'.\sigma_\kappa$$

$$\begin{array}{ll}
\Gamma \vdash b : \text{bool}_\kappa & \Gamma \vdash \text{default} : \tau_\kappa \to \tau_{\kappa'} \to \tau_{\kappa \vee \kappa'} \\
\Gamma \vdash \text{pre} : \tau_\kappa \to \tau_\kappa \to \tau_\kappa & \Gamma \vdash \text{reset} : \tau_\kappa \to \tau_{\kappa'} \to \tau_{\kappa \vee \kappa' \vee \kappa''} \\
\Gamma \vdash \text{sync} : \tau_\kappa \to \tau'_\kappa \to \text{unit}_\kappa & \exists!\delta, \Gamma \vdash \text{when} : \text{bool}_{\delta + \kappa} \to \tau_{\kappa'} \to \tau_{\delta \times \kappa'}
\end{array}$$

Fig. 7. Inference system $\Delta, \Gamma \vdash e : \tau$

Example 2. How do these considerations translate in the example of the timer? Since count, tick are synchronous, they share the same clock κ'. This clock is down-sampled in order to produce timeout signals. Hence, it it composed of an abstract clock δ and of a disjoint sub-clock κ, i.e., $\kappa' = \delta + \kappa$.

$$\begin{array}{ll}
\text{let count} = (\text{start when } (\text{last} \leq 1)) \text{ default } (\text{last} - 1) & \text{int}_{\delta + \kappa} \\
\mid \quad \text{last} = \text{pre count start} \mid \text{sync}(\text{count}, \text{tick}) & \text{int}_{\delta + \kappa} \\
\text{in when } (\text{last} \leq 1) \text{ true} & \text{bool}_\delta
\end{array}$$

What is the polymorphic type of the timer process? It has parameters start and tick which we said are of clock $\delta + \kappa$. The result is of clock δ. Since both δ and κ are context-free, we can generalize them (i.e. by considering $\kappa = \delta'$). In each place the timer is used (e.g. for sampling seconds and minutes in the watch), its existentially quantified clock receives a fresh instance, in order to distinguish it from other clocks. In the case of the watch, we actually have $\kappa_1 = \kappa_2 + \delta_2$.

$$
\begin{array}{rl}
\text{timer :} & \exists!\delta, \forall\delta'.(\text{int}_{\delta+\delta'} \rightarrow (\text{bool}_{\delta+\delta'} \rightarrow \text{bool}_\delta)_{\delta+\delta'})_{\kappa''} \\
\delta_1 \vdash \text{timer}(60) : & (\text{bool}_{\delta_1+\kappa_1} \rightarrow \text{bool}_{\delta_1})_{\delta_1+\kappa_1} \\
\delta_1, \delta_2 \vdash \text{timer}(100) : & (\text{bool}_{\delta_2+\kappa_2} \rightarrow \text{bool}_{\delta_2})_{\delta_2+\kappa_2}
\end{array}
$$

4 Causal Invariants

The clock calculus allows the compilation of parallelism within a process by defining a hierarchy of tasks according to clock relations [1] (see section 5). To generate executable code, additional information on the causal dependencies between signals is needed in order to serialize the emission of outputs in each task. In SIGNAL, this information is obtained by a flow analysis [2]. This analysis detects cyclic causal dependencies, which may incur dead-locks at runtime. Similarly, higher-order processes may define "instantaneously" recursive processes. Just as cyclic causal dependencies between first-order signals may denote dead-locks, cyclic higher-order processes may denote non-terminating processes. We address the issue of determining causal relation between signals by generalizing the flow analysis of SIGNAL in terms of an effect system (e.g., [18]) which relates each signal to the set of signals ϕ required for its computation (e.g. in $x = y + z$, x requires y and z).

Causality Inference A causality flow ϕ is either a variable φ (which identifies a signal) or an assembly φ, ϕ (which identifies signals ϕ that φ depends on). Flows are right-idempotent (i.e. $\varphi, \phi, \phi = \varphi, \phi$) and right-commutative (i.e. $\varphi, \phi, \phi' = \varphi, \phi', \phi$). A substitution $[\phi/\varphi]$ is defined by $(\varphi', \phi')[\phi/\varphi] = (\varphi'[\phi/\varphi]), (\phi'[\phi/\varphi])$. We reuse the symbols τ, σ and Γ to name flow-annotated types and environments. We write $\overline{\Gamma}(\tau^\phi)$ for the generalization of a type τ^ϕ over its set of free type and flow variables.

$$
\phi ::= \varphi \mid \varphi, \phi \quad \text{flow} \qquad \tau ::= \ldots \mid \tau^\phi \quad \text{type} \qquad \sigma ::= \ldots \mid \forall\varphi.\sigma \quad \text{scheme}
$$

Fig. 8. Types and causalities τ^ϕ

$$
\overline{\Gamma}(\tau^\phi) = \forall\alpha_{1..m}.\forall\varphi_{1..n}.\tau^\phi \ where \ \alpha_{1..m} = \text{ftv}(\tau)\backslash\text{ftv}(\Gamma) \ and \ \varphi_{1..n} = \text{ffv}(\tau)\backslash\text{ffv}(\Gamma)\backslash\text{ffv}(\phi)
$$

Fig. 9. Generalization $\overline{\Gamma}(\tau^\phi)$

The sequent $\Gamma \Vdash e : \tau$ checks that the flow hypothesis Γ is consistent with the assertion that e has flow and type τ. The rule (var) allows the instantiation of the polymorphic flow type $\Gamma(x)$ of a signal x by a monomorphic flow type τ. In the rule (par), the expressions e and e' may have different flows. The rules (let), (abs) and (app) are structurally equivalent to those of the clock calculus.

We write $(\tau^\phi) \times \phi' = \tau^{\phi,\phi'}$ and $(\tau_{1..n}) \times \phi = (\tau_1 \times \phi) \times \ldots (\tau_n \times \phi)$. The type of pre denotes no causality from the inputs (flows ϕ and ϕ') to the output ϕ''. The type of when denotes a causal dependency from the second input (flow ϕ) to the output (flow ϕ'', ϕ). The type of default denotes a causality from the inputs (ϕ and ϕ') to the output (ϕ'', ϕ, ϕ').

$$\frac{\begin{array}{c} \Gamma \uplus (x, \tau^\phi) \Vdash e : \tau^\phi \\ \Gamma \uplus (x, \overline{\Gamma}(\tau^\phi)) \Vdash e' : \tau' \\ \hline \Gamma \Vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau' \end{array}} \quad \text{(let)} \quad \frac{\begin{array}{c} \Gamma \Vdash e : (\tau' \to \tau)^\phi \\ \Gamma \Vdash e' : \tau' \\ \hline \Gamma \Vdash e(e') : \tau \times \phi \end{array}} \quad \text{(app)} \quad \frac{\begin{array}{c} \Gamma \Vdash e : \tau \\ \Gamma \Vdash e' : \tau' \\ \hline \Gamma \Vdash e \mid e' : \tau \times \tau' \end{array}} \quad \text{(par)}$$

$$\frac{\tau \preceq \Gamma(x)}{\Gamma \Vdash x : \tau} \quad \text{(var)} \quad \frac{\Gamma \Vdash e : \tau \quad \Gamma \Vdash e' : \tau' \quad \Gamma \Vdash p : \tau \to \tau' \to \tau''}{\Gamma \Vdash p\ e\ e' : \tau''} \quad \text{(pvs)}$$

$$\frac{\Gamma \uplus (x, \tau') \Vdash e : \tau}{\Gamma \Vdash \mathsf{proc}\ x\ e : (\tau' \to \tau)^{\phi, \phi'}} \quad \text{(abs)} \quad s.t.\ \phi' = \bigcup_{(y, \sigma^{\phi''}) \in \Gamma}^{y \in fv(e) \backslash \{x\}} \phi''$$

$$\Gamma \Vdash \mathsf{bool}^\phi \qquad \Gamma \Vdash \mathsf{sync} : \tau^\phi \to \tau'^{\phi'} \to \mathsf{unit}^{\phi''} \quad \Gamma \Vdash \mathsf{default} : \tau^\phi \to \tau^{\phi'} \to \tau^{\phi'', \phi, \phi'}$$

$$\Gamma \Vdash \mathsf{pre} : \tau^\phi \to \tau^{\phi'} \to \tau^{\phi''} \quad \Gamma \Vdash \mathsf{reset} : \tau^\phi \to \tau^{\phi'} \to \tau^{\phi'', \phi, \phi'} \quad \Gamma \Vdash \mathsf{when} : \mathsf{bool}^{\phi'} \to \tau^\phi \to \tau^{\phi'', \phi}$$

Fig. 10. Inference system $\Gamma \Vdash e : \tau$

Example 3. The information inferred by the causality calculus is depicted below for the process timer. Each signal is assigned a record that contains the flow variable that identifies it (form φ_1 to φ_5) and the flow of signals required for its computation. A linear checking of the equations in the process allows to infer the needed information. For instance count (flow φ_3) requires start (flow φ_1) or last (flow φ_4). Hence, its flow is $\varphi_3, \varphi_1, \varphi_4$. The signature of timer in the flow calculus is $\mathsf{int}^{\varphi_1} \to \mathsf{bool}^{\varphi_2} \to \mathsf{bool}^{\varphi_5}$.

$$\begin{bmatrix} \mathsf{start} : \mathsf{int}^{\varphi_1} \\ \mathsf{tick} : \mathsf{bool}^{\varphi_2} \end{bmatrix} \quad \Vdash \quad \begin{array}{ll} \mathsf{let}\ \mathsf{count} = (\mathsf{start}\ \mathsf{when}\ (\mathsf{last} \leq 1))\ \mathsf{default}\ (\mathsf{last} - 1) & \mathsf{int}^{\varphi_3, \varphi_1, \varphi_4} \\ \mid\ \ \mathsf{last} = \mathsf{pre}\ \mathsf{count}\ \mathsf{start} \mid \mathsf{sync}(\mathsf{count}, \mathsf{tick}) & \mathsf{int}^{\varphi_4} \\ \mathsf{in}\ \ \mathsf{when}\ (\mathsf{last} \leq 1)\ \mathsf{true} & \mathsf{bool}^{\varphi_5} \end{array}$$

5 Formal Properties

Typing rules are now required for reasoning on evaluated expressions r: axioms to mean that the absence has any type, clock and flow and rules for sequences.

$$\Delta, \Gamma \vdash \mathsf{abs} : \tau_\kappa \qquad \Gamma \Vdash \mathsf{abs} : \tau^\phi \qquad \frac{\Delta, \Gamma \vdash r : \tau \quad \Delta', \Gamma \vdash r' : \tau'}{\Delta \uplus \Delta', \Gamma \vdash (r, r') : \tau \times \tau'} \qquad \frac{\Gamma \Vdash r : \tau \quad \Gamma \Vdash r' : \tau'}{\Gamma \Vdash (r, r') : \tau \times \tau'}$$

Fig. 11. Typing rules $\Delta, \Gamma \vdash r : \tau$ and $\Gamma \Vdash r : \tau$ for results

Consistency We establish a consistency relation between environments E and hypothesis Δ, Γ to express the fact that clock relations model synchronizations and causality relations model data dependencies.

Definition 1 (Consistency). We write $\Delta, \Gamma \vdash E$ iff $dom(E) = dom(\Gamma)$ and for all $(x, r) \in E$ there exists $\exists \Delta'. \tau_\kappa \preceq \Gamma(x)$ such that $\Delta \supseteq \Delta'$, $\Delta \vdash v : \tau_\kappa$ and, for all $(x', r') \in E$ and $(x' : \sigma'_{\kappa'}) \in \Gamma$, $\kappa \geq \kappa'$ implies $(r = \mathsf{abs}\ \mathsf{iff}\ r' = \mathsf{abs})$. We write $\Gamma \Vdash E$ iff $dom(E) = dom(\Gamma)$ and for all $(x, r) \in E$ there exists $\tau^\phi \preceq \Gamma(x)$ such that $\Vdash v : \tau^\phi$ and, there exists $(x', r') \in E$ and $(x' : \sigma'^{\phi'}) \in \Gamma$ such that $\phi \supseteq \phi'$ implies $(r = \mathsf{abs}\ \mathsf{iff}\ r' = \mathsf{abs})$.

Invariance The theorem 1 states the invariance of temporal and causal properties determined by the inference systems with respect to a step in the operational semantics. It says that, if a system e is well-typed, given well-typed inputs E and executed, then the outputs $\tilde{r}$ and final state e' have same type.

Theorem 1. *If $\Delta, \Gamma \vdash e : \tau$, $\Delta, \Gamma \vdash E$ and $E \vdash e \xrightarrow{\tilde{x}} e'$ then $\Delta, \Gamma \vdash \tilde{r} : \tau$ and $\Delta, \Gamma \vdash e' : \tau$. If $\Gamma \Vdash e : \tau$, $\Gamma \Vdash E$ and $E \vdash e \xrightarrow{\tilde{x}} e'$ then $\Gamma \Vdash \tilde{r} : \tau$ and $\Gamma \Vdash e' : \tau$.*

The proof of theorem 1 requires a substitution lemma in which θ stands for a substitution of type variables α by types τ, of clock variables δ by clocks κ and of flow variables φ by flows ϕ.

Lemma 1. *If $\Delta, \Gamma \vdash e : \tau$ then $\Delta, \theta\Gamma \vdash e : \theta\tau$ for all θ s.t. $\Delta = \theta\Delta$. If $\Gamma \Vdash e : \tau$ then $\theta\Gamma \Vdash e : \theta\tau$ for all θ.*

Endochrony Endochrony refers to the Ancient Greek: "$\varepsilon\nu\delta o$", and literally means "time defined from the inside". An endochronous expression e defines a reactive system where "time defined from the inside" translates into the property that the production of its outputs only depends on the presence of its inputs. An endochronous system reacts to inputs (by having clocks computable from that of its inputs) and terminates within a predictable amount of time (by avoiding deadlocks and recursions). Hierarchization is the implementation of endochrony. It is the medium used in SIGNAL for compiling parallelism [1]. It consists of organizing the environment Γ as a tree Θ that defines a correct scheduling of computations into tasks. Each node of the tree consists of synchronous signals. It denotes the task of computing them when the clock is active. Each relation of a node with a sub-tree represents a sub-task of smaller clock. A tree of tasks being computed, expressions in each task can be serialized by following the causal relations determined by the flow analysis.

Example 4. A good example of exochronous specification is the expression count $= (\text{start when } (\text{last} \leq 1)) \text{ default } (\text{last} - 1)$ of the timer. Out of its context, it may be possible to give the signal count clock $\kappa_c = (\kappa_s) \vee (\kappa + \delta)$ given $\kappa_l = \kappa + \delta$ the clock of last. Endochrony is achieved by synchronizing count with start and last (with the pre statement) and count with the input tick (the sync statement). This hierarchizes the output of the timer w.r.t. the input tick and all local signals.

We express the property of hierarchization as a relation $\mathcal{H}_\Delta(\Gamma, \kappa)$ between the clock κ of an expression e and its environment Δ, Γ.

Definition 2 (Hierarchization). The clock κ is hierarchizable in Δ, Γ, written $\mathcal{H}_\Delta(\Gamma, \kappa)$ iff $\kappa \neq 0$, $\kappa \neq 1$ and there exists $(x, \sigma_{\kappa'}) \in \Gamma$ such that $\kappa \leq \kappa'$ and $\kappa' \not\leq \Sigma_{\delta \in \Delta}\delta$. We write $\mathcal{H}_\Delta(\Gamma, \tau)$ iff all clocks κ occurring free in τ satisfy $\mathcal{H}_\Delta(\Gamma, \kappa)$.

The property of deadlock-freedom $\mathcal{A}$ resources to flow annotations. It means that a signal $(x, \sigma^{\varphi, \phi}) \in \Gamma$ s.t. $\varphi \in \phi$ may be defined by a fixed-point equation. This means that its computation may dead-lock (e.g. let $x = x + 1$) or may not terminate (e.g. let $f = \text{proc } x \, f(x)$).

Definition 3 (Cycle-Freedom). The flow φ, ϕ is acyclic, written $\mathcal{A}(\varphi, \phi)$, iff $\varphi \notin \phi$. We write $\mathcal{A}(\tau)$ iff all flows ϕ occurring in τ satisfy $\mathcal{A}(\phi)$.

Given $\mathcal{H}$ and $\mathcal{A}$, we define endochrony as an inductive and decidable property.

Definition 4 (Endochrony). A judgments satisfy $\mathcal{E}[\![\Gamma \Vdash e : \tau]\!]$ iff $\mathcal{A}(\tau)$. A proof tree $P/\Gamma \Vdash e : \tau$ (resp. $P'/\Delta, \Gamma \vdash e : \tau$) is endochronous iff P (resp. P') is endochronous and $\mathcal{E}[\![\Gamma \Vdash e : \tau]\!]$ (resp. $\mathcal{E}[\![\Delta, \Gamma \vdash e : \tau]\!]$). A system e is endochronous iff it has endochronous proofs $P/\Gamma \Vdash e : \tau$ and $P'/\Delta, \Gamma \vdash e : \tau$.

$\mathcal{E}[\![\Delta, \Gamma \vdash x : \tau]\!]$ iff $\mathcal{H}_\Delta(\Gamma, \tau)$ $\quad \mathcal{E}[\![\Delta \uplus \Delta', \Gamma \vdash e(e') : \tau]\!]$ iff $\mathcal{H}_\Delta(\Gamma, \tau)$

$\mathcal{E}[\![\Delta, \Gamma \vdash e \mid e' : \tau]\!]$ iff $\mathcal{H}_\Delta(\Gamma, \tau)$ $\quad \mathcal{E}[\![\Delta, \Gamma \vdash \mathsf{let}\, x = e \,\mathsf{in}\, e' : \tau']\!]$ iff $\Delta, \Gamma \uplus (x : \tau) \vdash e : \tau$

$\mathcal{E}[\![\Delta, \Gamma \vdash \mathsf{proc}\, x\, e : \tau]\!]$ iff $\mathcal{H}_\Delta(\Gamma^{\{x\} \cup fv(e)}, \tau)$ $\qquad\qquad$ *and* $\mathcal{H}_\Delta(\Gamma, \tau)$, $\mathcal{H}_\Delta(\Gamma, \tau')$

Fig. 12. Endochrony $\mathcal{E}[\![\Delta, \Gamma \Vdash e : \tau]\!]$

Example 5. An example of non-endochronous specification is $\mathsf{x} + (\mathsf{x} \,\mathsf{when}\, \mathsf{x} > 0)$. It requires $x : \exists \delta.\mathsf{int}_\delta$ to reflect that x is present iff $x > 0$ (i.e. at the abstract clock δ). If $x \leq 0$, the expression deadlocks. As in SIGNAL, our definition of endochrony allows to detect and rejects such a constrained specification.

Safety The theorem 1 and the property of endochrony characterize the expressions e which "cannot go wrong".

Theorem 2. *If E and e are endochronous (with $\Delta, \Gamma \vdash E$, $\Gamma \Vdash E$, $\Gamma \Vdash e : \tau$ and $\Delta, \Gamma \vdash e : \tau$) then $E \vdash e \xrightarrow{\tilde{r}} e'$ and $\tilde{r}$ and e' are endochronous.*

6 Implementation

We can easily derive an algorithm $\inf(\Delta', \Gamma)[\![e]\!] = (\Delta, \tau, \theta)$ from the inference system. It constructs a set of constants Δ, a type τ and a substitution θ on types and clock variables from an initial Δ', a set of type hypothesis Γ and an expression e. Whereas the clock calculus introduces clocks κ and types τ, the inference system introduces fresh variables δ and α. Whereas the clock calculus specifies type matches, the inference system determines equations $\tau' =^? \tau''$ to be unified. The equational problem $\tau' =^? \tau''$ is decomposed in the resolution of equations on data-types and on clocks. Unification of boolean clock equations is decidable and unitary (i.e. boolean equations have a unique most general unifier [12]). In inf, we refer to $\mathrm{mgu}_\Delta(\kappa =^? \kappa')$ as the boolean unification algorithm of [12] determining the most general unifier of the equation $\kappa =^? \kappa'$ with constants Δ. Similarly, causality information φ, ϕ is represented by a simple form of extensible record [16] of prefix φ and members ϕ, for which we know the resolution of equations $\phi =^? \phi'$ to be a decidable and unitary equational problem. A clever implementation of our type system would account for the simplification of clock and flow annotations by resourcing to eliminations rules for clocks: $\Delta, \Gamma \vdash e : \tau$ iff $\Delta \uplus \delta, \Gamma \vdash e : \tau$ and $\delta \notin \mathrm{fcv}(\Gamma, \tau)$; and for flows: $\Gamma \Vdash e : \tau^\phi$ iff $\Gamma \Vdash e : \tau^{\phi, \varphi}$ and $\varphi \notin \mathrm{left}(\Gamma, \tau, \phi)$ (s.t.$\mathrm{left}(\varphi, \phi) = \{\varphi\}$). Showing that the inference of types, clocks and flows reduces to solving a system of boolean and record equations demonstrates its decidability.

7 Related Work

Contemporary work demonstrated the practicality of higher-order programming languages for the design of circuits (e.g. LAVA [4], HAWK [13]). Our type-theoretical modeling of synchronous interaction would find an interesting application in serving as a medium for specifying behavioral abstractions of circuit components in such systems. The introduction of LUCID-synchrone [6] was a first step towards the generalization of the principles of synchronous programming to higher-order languages. The proposal of [6] is a model of synchronous recursive functions over lazy streams. Our programming model is co-iterative and implements exochronous operators (i.e. when and default, as in SIGNAL). The clock calculus of [6] uses symbolic types c annotated with expressions (c on e, $c \xrightarrow{x} c'$). The use dependent types limits the expression of decidable clock relations (i.e. $c \geq c$ on e), of polymorphism (e.g. for eliminating let-bound identifiers x), imposes restrictions for typing functions $\lambda x.e$ (i.e. the x must be universally quantifiable). As types contain expressions and expressions model state transitions, specifying invariants under subject reduction becomes sophisticated ([6, Theorem 1] states the preservation of typability under reduction). Our type system proposes a both simpler and more expressive representation of both the temporal and causal invariants of synchronous processes. Other approaches for characterizing reactivity are proposed in [10], where an intuitive and expressive numerical model of time is presented for determining length-preserving recursive list functions in a purely functional language. Other interpretations of reactivity, mainly consisting of a programming model, are the object-oriented approach of [5], the functional approach of [15], or the transposition of synchronous programming paradigms under the concept of concurrent constraints in [17], or the notion of continuous time employed in [7]. None of the frameworks considered in [6, 10, 7, 15, 17] characterize the property of *endochrony* for reactive systems.

8 Conclusions

Our main contribution is to transpose the notion of *endochrony*, as modelled in SIGNAL, in the context of a higher-order and typed synchronous programming model. We introduce an expressive type system for specifying the temporal and causal relations between events. This system generalizes previous studies on clock calculi [2] in terms of effect systems [18]. It implements an abstraction of the temporal invariants of signals using boolean expressions, called *clocks*. A subject reduction property (section 5, theorem 1) states the correctness of clock invariants. A property of *hierarchization* and of *deadlock-freedom* allows to decide a safety property (theorem 2) and to characterize the compilation of parallelism in a process as the organization of its expressions as a tree of serialized actions. Our approach generalizes the semantics of equational synchronous programming languages (such as LUSTRE or SIGNAL) for the support higher-order processes. It allows to express the dynamic configuration and the generic specification of reactive system components while preserving the hypotheses on the synchrony of communications and on the instantaneousness of computations.

References

1. T. P. Amagbegnon, L. Besnard, P. Le Guernic. Implementation of the data-flow synchronous language SIGNAL. In *Conference on Programming Language Design and Implementation*. ACM, 1995.
2. A. Benveniste, P. Le Guernic, C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. In *Science of Computer Programming*, v. 16, 1991.
3. G. Berry, G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. In *Science of Computer Programming*, v. 19, 1992.
4. P. Bjesse, K. Claessen, M. Sheeran. LAVA: hardware design in HASKELL. In *International Conference on Functional Programming*. ACM, 1998.
5. F. Boussinot. ICOBJ programming. *Technical report* n. 3028. INRIA, 1996.
6. P. Caspi, M. Pouzet. Synchronous Kahn networks. In *International Conference on Functional Programming*. ACM, 1996.
7. C. Elliott, P. Hudak. Functional reactive animation. In *proceedings of the International Conference on Functional Programming*. ACM, 1997.
8. C. Fournet, G. Gonthier. The reflexive CHAM and the join calculus. In *Symposium on Principles of Programming Languages*. ACM Press, 1996.
9. N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. The synchronous data-flow programming language LUSTRE. In *Proceedings of the* IEEE, v. 79(9). IEEE, 1991.
10. J. Hughes, L. Pareto, A. Sabry. Proving the correctness of reactive systems using sized types. In *Symposium on Principles of Programming Languages*. ACM, 1996.
11. T. Jensen. Clock analysis of synchronous dataflow programs. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, 1995.
12. U. Martin, T. Nipkow. Unification in boolean rings. In *Journal of Automated Reasoning*, v. 4. Kluwer Academic Press, 1988.
13. J. Matthews, B. Cook, J. Launchbury. Microprocessor specification in HAWK. In *International Conference on Computer Languages*. IEEE, 1998.
14. R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. In *Information and Computation*. Academic Press, 1992.
15. R. Pucella. Reactive programming in STANDARD ML. In *International Conference on Computer Languages*. IEEE, 1998.
16. D. Remi. Type inference for records in a natural extension of ML. *Research report n. 1431*. INRIA, 1991.
17. V. Saraswat, R. Jagadeesan, V. Gupta. Foundation of timed concurrent constraint programming. In *Symposium on Logic in Computer Science*. IEEE, 1994.
18. J.-P. Talpin, P. Jouvelot. The type and effect discipline. In *Information and Computation*, v. 111(2). Academic Press, 1994.