Reactive Types

Jean-Pierre Talpin

IRISA (INRIA-Rennes & CNRS URA 227)
Campus de Beaulieu, 35000 Rennes, France
E-mail: talpin@irisa.fr

Abstract. Synchronous languages, such as SIGNAL, are best suited for the design of dependable real-time systems. Synchronous languages enable a very high-level specification and an extremely modular implementation of complex systems by structurally decomposing them into elementary synchronous processes. Separate compilation in reactive languages is however made a difficult issue by global safety requirements.

We give a simple and effective account to the separate compilation of reactive systems by introducing a *specification as type* paradigm for reactive languages: *reactive types*. Just as data-types describe the structure of data in conventional languages, reactive types describe the structure of interaction in synchronous languages. We define an inference system for determining reactive types in the SIGNAL language and show how to reconstruct adequate compile-time information on reactive programs by means of such specifications.

1 Introduction

A reactive system is a computer system which continuously *reacts* to its environment. Many industrial systems are reactive in nature: process control systems, monitoring systems, signal processing systems, communication protocols. Reactive systems are commonly characterized by critical requirements such as fast reaction time or bounded memory usage.

Classical design tools for implementing reactive systems, such as real-time operating system or general-purpose concurrent languages (e.g. ADA), neither provide a global and formal view of the system (separated into tasks or services) nor preserve its determinism.

Synchronous languages, such as SIGNAL [3], LUSTRE [5] or ESTEREL [4], are specifically designed to ease the development of reactive systems by providing both a formal view and a logical notion of concurrency preserving determinism: *synchronous* concurrency, where operations and communications are instantaneous.

In a synchronous language, concurrency is meant as a way to logically decompose the description of a system into a set of elementary communicating processes. Interaction between concurrent components within the program is conceptually performed by broadcasting events.

In practice, a synchronous program is usually translated into a circuit or into a monolythic automaton. The hypothesis of synchrony is translated into the requirement that the program reacts rapidly to its environment.

As a result, synchronous languages allow a very high-level specification and an extremely modular implementation of complex systems by structurally decomposing their functional components into elementary processes. Although modularity is a key advantages of synchronous languages, separate compilation is made a difficult issue by the requirements of proving global safety properties of the system.

To enable separate compilation of the functional components of reactive systems while preserving their global integrity, we introduce the notion of *reactive type.* Just as data-types describe the structure of data in conventional languages, reactive types describe the structure of interaction in synchronous languages.

In conventional languages, function types are the media enabling separate compilation of procedures in a program. Similarly, reactive types can be used as a medium for separately compiling reactive processes and assembling them to form complex systems.

In this paper, we present an inference system which associates SIGNAL programs to reactive types. We prove its correctness w.r.t. the dynamic semantics of SIGNAL and show how to reconstruct adequate compile-time information on programs by means of reactive types.

2 An Overview of SIGNAL

SIGNAL is an equational synchronous programming language: a SIGNAL program is modularly organized into processes consisting of simultaneous equations on signals. In SIGNAL, an equation is an elementary and instantaneous operation on input signals which defines an output. A signal is a sequence of values defined over a totally ordered set of instants. At any given instant, a signal x is either present or absent (its clock $\hat{x}$ denotes the instants at which it is present).

Syntax A process p is either an elementary equation, the synchronous composition $p \mid p'$ of two processes p and p' or the declaration p/x of a local signal x in a process p. An equation instantaneously maps a signal x to a value v (e.g. an event, a boolean, an integer), to the previous value "$y\,\$1$" of a signal y, to a synchronous operation $f(y, z)$ on the signals y and z (e.g. an operation on booleans, on numbers), by merging two signals y and z or by sampling a signal y under a condition z.

$$\begin{array}{rll}
p ::= & (p \mid p') & \text{synchronous composition} \\
\mid & p/x & \text{encapsulation or scoping} \\
\mid & x := v & \text{constant declaration} \\
\mid & x := f(y, z) & \text{synchronous operation} \\
\mid & x := y \ \mathsf{when}\ z & \text{down-sampling} \\
\mid & x := y \ \mathsf{default}\ z & \text{deterministic merge} \\
\mid & x := y\,\$1\ \mathsf{init}\ v & \text{delay operation}
\end{array}$$

Fig. 1. Syntax of processes p in SIGNAL

Dynamic Semantics The dynamic semantics of SIGNAL, written $p \xrightarrow{e} p'$, presented in [3], outlined in the figure 2, describes how a process p evolves over time. A transition in the dynamic semantics defines an instant. Each instant is characterized by a set e of simultaneous events and by an instantaneous transition from a state p to a state p'. The environment e associates a signal x to a value v when it is present (written $x(v)$) or to $\perp$ when it is absent (written $x(\perp)$).

$$\frac{p \xrightarrow{e} p'' \quad p' \xrightarrow{e'} p''' \quad e \cap e'}{p \mid p' \xrightarrow{e \cup e'} p'' \mid p'''} \qquad \frac{p \xrightarrow{e\ x(v)} p'}{p/x \xrightarrow{e} p'/x}$$

$$\begin{array}{lcl|lcl}
x := v & \xrightarrow{x(v)} & x := v & x := y \text{ when } z & \xrightarrow{x(\perp)\ y(v)\ z(\perp)} & x := y \text{ when } z \\
x := f(y,z) & \xrightarrow{x(\perp)\ y(\perp)\ z(\perp)} & x := f(y,z) & x := y \text{ when } z & \xrightarrow{x(v)\ y(v)\ z(t)} & x := y \text{ when } z \\
x := f(y,z) & \xrightarrow{x(f(v,v'))\ y(v)\ z(v')} & x := f(y,z) & x := y \text{ when } z & \xrightarrow{x(\perp)\ y(v)\ z(f)} & x := y \text{ when } z \\
x := y\,\$1 \text{ init } v & \xrightarrow{x(\perp)\ y(\perp)} & x := y\,\$1 \text{ init } v & x := y \text{ default } z & \xrightarrow{x(\perp)\ y(\perp)\ z(\perp)} & x := y \text{ default } z \\
x := y\,\$1 \text{ init } v & \xrightarrow{x(v)\ y(v')} & x := y\,\$1 \text{ init } v' & x := y \text{ default } z & \xrightarrow{x(v)\ y(v)} & x := y \text{ default } z \\
x := y \text{ when } z & \xrightarrow{x(\perp)\ y(\perp)} & x := y \text{ when } z & x := y \text{ default } z & \xrightarrow{x(v)\ y(\perp)\ z(v)} & x := y \text{ default } z
\end{array}$$

Fig. 2. Dynamic semantics of SIGNAL

Parallel composition $p \mid p'$ synchronizes the events e and e' produced by p and p'. The relation $e \cap e'$ is defined iff $e(x) = e'(x)$ for all x in $dom(e) \cap dom(e')$).

A synchronous operation $x := f(y,z)$ instantaneously computes the value of f by y and z and outputs it to the signal x. A delay $x := y\,\$1 \text{ init } v$ stores the value v' of y and outputs the previous value v to x.

A merge $x := y \text{ default } z$ outputs the value of y to x when y is present or the value of z otherwise. A sampling $x := y \text{ when } z$ outputs y to x when z is present and true. When all the inputs of an equation are absent, a transition takes place but no value is given to its output.

Example 1. As a first example, we consider the stream of positive integers nat as the synchronous composition of two equations. The first equation defines the local signal y initially as 0 and then as the previous value of x. The second equation defines x as y plus 1.

```
process nat (out integer x) =
        ( y := x $ 1 init 0
        | x := y + 1
        ) / y
```

At each instant n, both equations are executed simultaneously (this explains why x_n is defined by $x_{n-1} + 1$ and not by $x := x + 1$ as in a conventional programming language). Notice that the rate of the execution of the program is not constrained by an external input signal. The schedule of its execution will actually depend on the use of x in the environment.

Data-Flow Graphs As outlined in the previous example, the compilation of SIGNAL requires the static resolution of temporal relations between signals. In order to ensure the respect of synchronization constraints expressed in programs.

$$\begin{array}{ll} c ::= \widehat{x} \mid \widehat{x}\backslash\widehat{y} \mid [x] \mid c \wedge c \mid c \vee c & \text{clock} \\ g ::= \emptyset \mid (\widehat{x} = c) \mid (x \xrightarrow{c} y) \mid g \cup g' \mid \exists x.g & \text{graph} \end{array}$$

Fig. 3. Clocks c and conditional data-flow graphs g

The control model of a SIGNAL program is represented by a set of temporal relations $\widehat{x} = c$ between signal clocks $\widehat{x}$ and expression clocks c. The clock $\widehat{x}$ denotes the instants when x is present. The clock $[x]$ denotes the presence of a boolean signal x with the value true. The clock $\widehat{x}\backslash\widehat{y}$ denotes the instants where x is present and y absent. The formula $c \wedge c'$ and $c \vee c'$ denote the conjunction and disjunction of the instants c and c'.

The data-flow model of a program is represented by a graph composed of arrows $x \xrightarrow{c} y$. An arrow $x \xrightarrow{c} y$ denotes a dependency from x to y at the clock c.

References to local signals x in a graph g are bound by existential quantification $\exists x.g$. We write $fv(g)$ and $bv(g)$ for the free and bound signals of g. We write $\exists y.g = \exists x.(g[x/y])$ and $(\exists x.g') \cup g = \exists x.(g \cup g')$ iff $x \notin fv(g) \cup bv(g)$. We identify $\exists y.(\exists x.g)$ and $\exists x.(\exists y.g)$.

Clock Calculus The analysis which determines the conditional data-flow graph g of a SIGNAL program is called the clock calculus (introduced in [3], outlined in the figure 4). Using the graph g produced by clock calculus, the SIGNAL compiler generates an optimal compile-time scheduling of the actions specified in the source program by hierarchizing the temporal relations in g and by ruling the execution of the program using its master clock [2]. Using the graph g, causal dependencies in the source program can easily be detected as constrained boolean conditions on signals (e.g. $[x] = \hat{y}$) or cyclic data dependencies (e.g. $x \xrightarrow{c} x$).

$$x := v \Rightarrow \emptyset \qquad x := y\,\$1 \Rightarrow (\hat{x} = \hat{y}) \qquad x := y \text{ when } z \Rightarrow \begin{pmatrix} y \xrightarrow{[z]} x \\ \hat{x} = \hat{y} \wedge [z] \end{pmatrix}$$

$$x := f(y,z) \Rightarrow \begin{pmatrix} y \xrightarrow{\hat{x}} x, z \xrightarrow{\hat{x}} x \\ \hat{x} = \hat{y}, \hat{x} = \hat{z} \end{pmatrix} \qquad x := y \text{ default } z \Rightarrow \begin{pmatrix} y \xrightarrow{\hat{y}} x, z \xrightarrow{\hat{z}\backslash\hat{y}} x \\ \hat{x} = \hat{y} \vee \hat{z} \end{pmatrix}$$

$$\frac{p \Rightarrow g}{p/x \Rightarrow \exists x.g} \qquad \frac{p \Rightarrow g \quad p' \Rightarrow g'}{p \mid p' \Rightarrow g \cup g'}$$

Fig. 4. Clock calculus $p \Rightarrow g$

Example 2. In the case of the process nat, for instance, the clock calculus determines the temporal relation between x and y: $\widehat{x} = \widehat{y}$ (i.e. x and y are synchronous) and the data dependency $y \xrightarrow{\hat{x}} x$ (i.e. the computation of x depends on the value of y). Notice that no external constraint exists on the signal x.

```
process nat (out integer x) =
        ( y := x $ 1 init 0
        | x := y + 1
        ) / y
```

$$\hat{x} = \hat{y} \qquad y \rightarrow x \qquad \exists y$$

Separate Compilation As the conditional data-flow graph of a SIGNAL program is the essential medium for checking its safety and compiling it, separate compilation is made a difficult issue by the requirements of proving global safety properties. To illustrate this issue, let us consider a typical situation raised in the following process definition. Let copy be the process which assigns the value of its input signals a and b to its output signals x and y: (x:=a | y:=b).

```
process copy (in a, b; out x, y) =
        ( x := a
        | y := b
        )
```

$$a \rightarrow x \qquad b \rightarrow y$$

To compile it, the SIGNAL compiler has the choice between scheduling either x:=a; y:=b or y:=b; x:=a. However, the appropriate choice depends on the way the process is invoked. For instance, in the case below, only the first scheduling (i.e. u:=w; v:=u) is correct (the second dead-locks).

```
(u,v) := copy(w,u)
```

$$w \rightarrow u \rightarrow v$$

Fortunately, this problem can be solved by determining the data dependencies $a \rightarrow x$ and $b \rightarrow y$ between (a,b) and (x,y) where copy is defined and the actual data dependencies $w \rightarrow u$ and $u \rightarrow v$ between u, v and w where copy is used. In a situation of separate compilation that information is however not directly accessible by the textual definition of the process copy. In this case, the explicit declaration of the temporal and data-flow relations between signals appears to be necessary for solving the issue of separate compilation.

3 Reactive Types

A common justification of typing in conventional programming languages is that "*well-typed programs do not go wrong*" [Milner]. Our reactive type system provides simple and effective means for making similar statements on reactive systems. Just as data-types abstract the representation of data, reactive types abstract the interaction structure of processes. The definition of our reactive type system incorporates both a calculus for reasoning on the interaction structure of SIGNAL programs and a way to check basic safety requirements.

In the type system of the figure 5, a signal x is an atomic reactive type. A constant has type $\bot$. A delay operation on a signal of type t has type δt. The down-sampling of a signal of type t by a signal of type t' has type $t \ominus t'$. The deterministic merge between two signals of types t and t' has type $t \oplus t'$. An equation of input type t and of output signal x has type $t \rightarrow x$. The recursive (resp. local) definition of x in t is written $\rho x.t$ (resp. $\exists x.t$). The synchronization of two signals of types t and t' is written $t \times t'$, its composition $t \otimes t'$.

$t ::= \bot$	constant		$\mid t \rightarrow x$	equation
$\mid\ x$	signal		$\mid \rho x.t$	recursion
$\mid\ \delta t$	delay		$\mid \exists x.t$	scoping
$\mid\ t \ominus t'$	sample		$\mid t \times t'$	synchronization
$\mid\ t \oplus t'$	merge		$\mid t \otimes t'$	composition

Fig. 5. Reactive types t

In the figure 6, we define the inference system for reactive types. It is written $p{:}t$ and associates a SIGNAL process p to its reactive type t.

$$x := v : \bot \rightarrow x$$
$$x := y\,\$1 : (\delta y) \rightarrow x$$
$$x := y \text{ when } z : (y \ominus z) \rightarrow x$$
$$x := y \text{ default } z : (y \oplus z) \rightarrow x$$
$$x := f(y,z) : (x \times (y \times z)) \otimes ((y \otimes z) \rightarrow x)$$

$$\frac{p{:}t \quad p'{:}t'}{p \mid p'{:}t \otimes t'} \qquad \frac{p{:}t}{p/x{:}\exists x.t}$$

Fig. 6. Inference system $p{:}t$

In order to identify reactive types which denote identical interaction structures (formally, in the sense of definition 2 and of theorem 3), we equip our type system with algebraic rules. Types constructed with δ, $\otimes$ and $\times$ are sets with neutral element $\bot$ and satisfy the distribution rules of the figure 7.

$$\delta\bot = \bot \qquad \delta(\delta t) = \delta t \qquad \delta(\rho x.t) = \rho x.(\delta t)$$
$$\delta(t \ominus t') = (\delta t) \ominus (\delta t') \qquad \delta(t \oplus t') = (\delta t) \oplus (\delta t') \qquad \delta(t \otimes t') = (\delta t) \otimes (\delta t')$$

$$t \times (t' \times \bot) = t \times (t' \times t') = t \times t' \qquad t \otimes (t' \times \bot) = t \otimes (t' \times t') = t$$
$$(t \times (t' \otimes t'')) = (t \times (t' \times t'')) \qquad (t \times (t' \times t'')) = (t \times t'') \otimes (t' \times t'') \otimes (t \times t'')$$

$$t \oplus (t' \otimes t'') = (t \oplus t') \otimes (t \oplus t'') \qquad (t' \otimes t'') \oplus t = (t' \oplus t) \otimes (t'' \oplus t) \qquad t \oplus t = t$$
$$t \ominus (t' \otimes t'') = (t \ominus t') \otimes (t \ominus t'') \qquad (t' \otimes t'') \ominus t = (t' \ominus t) \otimes (t'' \ominus t) \qquad \bot \ominus \bot = \bot$$

Fig. 7. Distribution of δt, $t \otimes t'$ and $t \times t'$

The signal x referenced in a type $\rho x.t$ or $\exists x.t$ is bound to t. Bound signals x satisfy the scoping rules of the figure 8. We write $fv(t)$ (resp. $bv(t)$) for the set of free (resp. bound) signals of a type t. We write $t_x = t \otimes_{((t' \times x) \otimes (t'' \times x)) \in t} (t' \times t'')$ for the completion of t w.r.t. x [to show, e.g., that $y \times z = \exists x.((\bot \rightarrow x) \otimes (x \times y) \otimes (x \times z))$].

$$\rho x.t = t[\rho x.t/x] \qquad \frac{x \notin fv(t)}{\rho x.t = t} \qquad \frac{x \notin fv(t)}{\rho y.t = \rho x.(t[x/y])} \qquad \frac{x \notin fv(t)}{t \otimes (\rho x.t') = \rho x.(t \otimes t')}$$

$$\exists x.(t \otimes (t' \rightarrow x)) = t_x[\rho x.t'/x] \qquad \frac{x \notin fv(t)}{\exists x.t = t} \qquad \frac{x \notin fv(t)}{\exists y.t = \exists x.(t[x/y])} \qquad \frac{x \notin fv(t)}{t \otimes (\exists x.t') = \exists x.(t \otimes t')}$$

Fig. 8. Scoping properties of reactive types t

The algebraic properties of reactive types give rise to the definition of a normal form (definition 1). Every type t obtained from the inference system $p{:}t$ of the figure 6 can be represented in normal form.

Definition 1. The normal form of a reactive type is $t ::= t'' \times t'' \mid t' \rightarrow x \mid t \otimes t$ where t' and t'' are of the form $t' ::= \bot \mid t'' \otimes t''$ and $t'' ::= x \mid \delta x \mid t'' \ominus t'' \mid t'' \oplus t'' \mid \rho x.t''$.

Example 3. To demonstrate the use of our reactive type system, let us consider a definition of the explicit synchronization of two signals x and y in SIGNAL.

```
process synchro (in x; out y) =
          ( z := ((x = x) = (y = y))
          ) / z
```

$$\begin{array}{l} (x \times y) \times z \\ (x \otimes y) \rightarrow z \\ \exists z \end{array}$$

The inference system of the figure 6 determines the interaction structure of the process synchro: $\exists z.((x \times y) \times z) \otimes ((x \otimes y) \rightarrow z))$. The algebraic rules of our reactive type system allow us to define its normal form as $\exists z.(((x \times y) \times z) \otimes ((x \otimes y) \rightarrow z)) = (x \times y) \times (\rho z.(x \otimes y)) = (x \times y) \times (x \otimes y) = x \times y$.

Safety The safety of a process p of reactive type t can be checked using the function $\mathcal{A}$. $\mathcal{A}_x(t)$ checks that their is no cycles from the input type t and the output signal x of an equation. The function $\mathcal{A}(t)$ checks for the absence of cycles in t. The function $\mathcal{C}(t)$ checks that the synchronizations in t do not incur causal dependencies.

$$\mathcal{A}_x(\bot) \quad \mathcal{A}_x(\delta t) \qquad \frac{x \neq y}{\mathcal{A}_x(y)} \qquad \frac{\mathcal{A}_x(t)}{\mathcal{A}_x(t \ominus t')} \qquad \frac{\mathcal{A}_x(t) \quad \mathcal{A}_x(t')}{\mathcal{A}_x(t \oplus t')} \qquad \frac{\mathcal{A}_x(t) \quad \mathcal{A}_x(t')}{\mathcal{A}_x(t \otimes t')} \qquad \frac{\mathcal{A}_x(t) \quad \mathcal{A}_y(t)}{\mathcal{A}_x(\rho y.t)}$$

$$\frac{\mathcal{A}(t)}{\mathcal{A}(\exists x.t)} \qquad \frac{\mathcal{A}_x(t) \quad \mathcal{A}(t'[t/x])}{\mathcal{A}((t \rightarrow x) \otimes t')} \qquad \frac{\mathcal{C}(t)}{\mathcal{A}(t)} \qquad \frac{\mathcal{C}(t \times t') \quad \mathcal{C}(t'')}{\mathcal{C}((t \times t') \otimes t'')} \qquad \frac{t' \neq t'' \ominus t}{\mathcal{C}(t \times t')}$$

Fig. 9. Property $\mathcal{A}(t)$

Example 4. To give an example of causal dependency and illustrate the combined use of our inference system and of the functions $\mathcal{A}$ and $\mathcal{C}$, we consider the process sample which computes a value w given two inputs u and v and a condition on them.

```
process sample (in integer u, v; out integer w) =
        ( x := u < v
        | y := u when x
        | w := v + y
        ) / logical x; integer y
```

The process sample uses two local signals x and y. Its reactive type declares the signals synchronous to w and the interaction structure leading to w:

$$((u \times v) \times w) \otimes ((v \otimes (u \ominus (u \otimes v))) \rightarrow w)$$

The function $\mathcal{C}$ detects that the process sample requires: $u \times (u \ominus u)$. By reconstructing the clock relations of the process sample (as shown in the figure 10), one observes that the constraints $\hat{u} = \hat{v}$ and $\hat{u} \wedge [x] = \hat{v}$ (where $\hat{x} = \hat{u} \vee \hat{v}$) cannot be satisfied simultaneously. This means that the process sample deadlocks if u and v are present and if the condition x is simultaneously false.

Graph Reconstruction In a conventional programming language, typing is a medium which enables the separate compilation of the functions in a program. Types allow to determine the representation of data in a program at compile-time. In a synchronous language, clock relations and data-flow dependencies allow to determine the appropriate scheduling of the actions specified as a system of simultaneous equations. This information can be reconstructed given the reactive type of a process.

$$\overrightarrow{[\![\bot]\!]}_y = \exists x.(\hat{x}, \emptyset) \qquad \overrightarrow{[\![x]\!]}_y = (\hat{x}, x \xrightarrow{\hat{x}} y) \qquad \overrightarrow{[\![\delta x]\!]}_y = (\hat{x}, \emptyset)$$

$$\frac{\overrightarrow{[\![t]\!]}_y = \exists x_{1..n}.(c, g)}{\overrightarrow{[\![\rho x.t]\!]}_y = \exists x, x_{1..n}.(c, (\hat{x} = c) \cup g \setminus g^x)} \qquad \frac{\overrightarrow{[\![t]\!]}_y = \exists x_{1..n}.(c, g) \quad \overrightarrow{[\![t']\!]}_y = \exists x_{n+1..m}.(c', g')}{\overrightarrow{[\![t \otimes t']\!]}_y = \exists x_{1..m}.(c \vee c', g \cup g')}$$

$$\frac{\overrightarrow{[\![t]\!]}_y = \exists x_{1..n}.(c, g) \quad \overrightarrow{[\![t']\!]}_y = \exists x_{n+1..m}.(c', g')}{\overrightarrow{[\![t \ominus t']\!]}_y = \exists x, x_{1..m}.(c \wedge [x], \hat{g} \cup \hat{g}' \cup (\hat{x} = c') \cup (z \xrightarrow{c'' \wedge [x]} y)_{z \xrightarrow{c''} y \in g})}$$

$$\frac{\overrightarrow{[\![t]\!]}_y = \exists x_{1..n}.(c, g) \quad \overrightarrow{[\![t']\!]}_y = \exists x_{n+1..m}.(c', g')}{\overrightarrow{[\![t \oplus t']\!]}_y = \exists x_{1..m}.(c \vee c', g \cup \hat{g}' \cup (z \xrightarrow{c'' \setminus c} y)_{z \xrightarrow{c''} y \in g'})} \qquad \frac{\vec{t} = g \quad \vec{t'} = g'}{\overrightarrow{t \otimes t'} = g \cup g'}$$

$$\frac{\overrightarrow{[\![t]\!]}_x = \exists x_{1..n}.(c, g) \quad \overrightarrow{[\![t']\!]}_x = \exists x_{n+1..m}.(c', g')}{\overrightarrow{t \times t'} = \exists x.(\hat{x} = c, \hat{x} = c') \cup \hat{g} \cup \hat{g}'} \qquad \frac{\overrightarrow{[\![t]\!]}_x = \exists x_{1..n}.(c, g)}{\overrightarrow{t \rightarrow x} = (\hat{x} = c) \cup g}$$

Fig. 10. Graph of a reactive type t in normal form

In the figure 10, the term $\overrightarrow{[\![t]\!]}_x = \exists x_{1..n}.(c, g)$ associated to an expression type t consists of a sequence of bound signals $x_{1..n}$ (introduced during the reconstruction), of a clock c (the clock of the expression) and of a graph g. We write $\overrightarrow{t}$ for the conditional data-flow graph reconstructed from the reactive type t of a SIGNAL process. We write $\hat{g}$ for the set of clock equations $\hat{x} = c$ in a graph g. We write $g^x = \{x \xrightarrow{c} y \in g\}$ and $g_x = \{y \xrightarrow{c} x \in g\}$.

4 Formal Properties

Adequacy We show that the reconstruction of the clock and data dependency relations $\overrightarrow{t}$ of a reactive type t is adequate with respect to the clock calculus of SIGNAL (figure 4) in that it is equivalent (in the sense of the definition 2) to the graph g inferred from a SIGNAL process.

Definition 2. $g \simeq g'$ iff there exists a renaming of $bv(g)$ and $bv(g')$ s.t. $\hat{g} \Leftrightarrow \hat{g'}$ and s.t. there exists $x \xrightarrow{c'} y \in \overline{g'}$ (resp. $\overline{g}$) s.t. $g \Rightarrow (c = c')$ for all $x \xrightarrow{c} y \in \overline{g}$ (resp. $\overline{g'}$).

In the above definition, we write $\hat{g} \Leftrightarrow \hat{g'}$ iff $\hat{g} \Rightarrow \hat{g'}$ and $\hat{g'} \Rightarrow \hat{g}$. We write $\hat{g} \Rightarrow \hat{g'}$ iff for all $(\hat{x} = c) \in \hat{g'}$, $g \Rightarrow (\hat{x} = c)$. We write $\overline{g}$ for the transitive closure $g|_{bv(g)}$ of the data-flow dependencies in the graph g w.r.t. its bound signals. We write $g|_x$ for the closure of the graph g with respect to x (notice that $g|_x$ does not necessarily eliminates all references to x, e.g., when $\hat{x} = c \in g$).

$$g|_x = ((g \backslash g_x) \backslash g^x) \cup (y \xrightarrow{c \wedge c'} z)_{(y \xrightarrow{c} x, x \xrightarrow{c'} z) \in g_x \times g^x}$$

The theorem 3 says that the conditional dependency graph $\overrightarrow{t}$ reconstructed from the type t of a process p is equivalent to the transitive closure of the graph g determined by the standard clock calculus of SIGNAL.

Theorem 3 (Adequacy). *If $p : t$ and $p \Rightarrow g$ then $g \simeq \overrightarrow{t}$.*

Proof. The proof is by induction on the structure of p using the rules defined in the figures 4 and 6. It uses the fact that the equivalence rules defined in the figure 7 and 8 preserve the definition 2 (i.e. if $t = t'$ then $\overrightarrow{t} \simeq \overrightarrow{t'}$).

Soundness The soundness of our reactive type system with respect to the dynamic semantics of SIGNAL is formulated as a subject reduction theorem. It says that the reactive type of a process is preserved during execution. We write "$e : t$ w.r.t. p" when the events e are consistent with the type t of a process p. We write $out(p)$ for the output signals of p and $in(p)$ for $fv(p) \setminus out(p)$.

Definition 4. $e : t$ w.r.t. p iff, for all $x(v) \in e_{out(p)}$, $t' \to x \in t$ and, for all $x(v) \in e_{in(p)}$, there exists $(t' \times t'') \in t$ or $t' \to y \in t$ s.t. $x \in fv(t')$.

Theorem 5 (Soundness). *If $p \xrightarrow{e} p'$ and $p : t$ then $e : t$ w.r.t. p and $p' : t$.*

Proof. The proof is by induction on the structure of p using the rules defined in the figures 2 and 6. It uses the fact that the equivalence rules defined in the figure 7 and 8 preserve the property of the definition 4.

Subtyping An account to the expressiveness of reactive types is that a notion of refinement can be introduced in our inference system in terms of a subtyping relation. Refinement is an important feature of the programming methodology of SIGNAL. It can be defined by the relation $g \sqsubseteq g'$ as follows.

Definition 6. For any graph g, $g \sqsubseteq g \cup (x \xrightarrow{c} y)$ and $g \sqsubseteq g \cup (\hat{x} = c)$ iff $g \cup (\hat{x} = c) \Rightarrow g$.

A graph g satisfies $g \sqsubseteq g'$ if it has less clock constraints and less data dependencies than g'. By inducing less clock and data dependencies, the specification g preserves the safety requirements related to the specification g'. In particular, it ensures that no boolean condition $[x]$ in g' is constrained and that no data dependency in g' is turned into a causal dependency (i.e a cycle $x \xrightarrow{c} x$).

In our inference system, subtyping can be introduced by means of an additional rule: if the process p has type t and $t \sqsubseteq t'$, then p also has type t'. The subtyping relation is defined by:

Definition 7. $t \sqsubseteq t'$ iff there exists t'' s.t. $t' = t \otimes t''$ and $\mathcal{A}(t')$.

Example 5. To illustrate the use of subtyping, let us reconsider the example of the process copy. The definition of copy has type $t = (a \rightarrow x) \otimes (b \rightarrow y)$ and its use has type $t' = (w \rightarrow u) \otimes (w \rightarrow v)$.

```
process copy (in a, b; out x, y) = (x:=a | y:=b)
(u,v) := copy(w,u)
```
$(a \rightarrow x) \otimes (b \rightarrow y)$
$(w \rightarrow u) \otimes (u \rightarrow v)$

In order to separately compile the process copy in such a way to support the scheduling required by its usage in the statement (u,v) := copy(w,u) of the program, one may use the subtyping relation $t' \sqsubseteq t''$ in order to give the type $t'' = t' \otimes (x \rightarrow b)$ explicitly to the definition of copy. This would enforce copy to be compiled as x:=a; y:=b and to be compatible with its use. To probe further by making an analogy to data-types, the process copy can be given polymorphic type $\forall \alpha, \alpha'.(\alpha \times \alpha') \rightarrow (\alpha \times \alpha')$ (using a polymorphic type inference algorithm such as that presented in [9]). Similarly, the explicit assignment of the "*less general*" type (int × bool) → (int × bool) to copy enforces x and a (resp. y and b) to be represented as integers (resp. boolean) by the SIGNAL compiler and to be used as such in the rest of the program.

5 Related Work

The definition of type systems for describing interaction in synchronous programming languages has been the subject of recent investigations. In [6], T. Jensen gives a model of SIGNAL using abstract interpretation and shows how to derive the clock calculus of [3] from this interpretation. In [1], S. Abramsky & al. give a categorical model of synchronous interaction in SIGNAL and LUSTRE and propose a related type system. In contrast to reactive types both type systems do not satisfy an adequacy theorem (in the sense of theorem 3): they do not permit to reconstruct all the compile-time information the SIGNAL compiler requires. Nonetheless, we believe that reactive types could be given an interpretation in the interaction category.

6 Conclusion

We have introduced a notion of *reactive type* for synchronous languages. Just as data-types describe the structure of data in conventional languages, reactive types describe the structure of interaction in reactive languages. We have introduced an inference system to associate SIGNAL programs to reactive types in the same way types are associated to functions in the lambda-calculus. Using reactive types, we have shown how to reconstruct the information needed for compiling SIGNAL programs and stated the correctness of our inference system with respect to the dynamic semantics of SIGNAL. We have introduced a notion of subtyping, which allows the gradual specification of SIGNAL programs using reactive types. Although our presentation was focused on SIGNAL, we believe that reactive types could equally be used in other synchronous languages, such as LUSTRE or ESTEREL, to type synchronous interaction.

References

1. S. Abramsky, S. Gay and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In *Proceedings of the 1994 Marktoberdorf Summer School.* NATO ASI Series, Springer Verlag, 1995.
2. T. P. Amagbegnon, L. Besnard and P. Le Guernic. Implementation of the data-flow synchronous language SIGNAL. In *Proceedings of the 1995's ACM Conference on Programming Language Design and Implementation*, p. 163–173. ACM, 1995.
3. A. Benveniste, P. Le Guernic and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.
4. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. In *Science of Computer Programming*, 19:87–152, 1992.
5. N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud. The synchronous data-flow programming language LUSTRE. In *Proceedings of the IEEE*, 79(9), p. 1305–1320. IEEE Press, September 1991.
6. T. Jensen. Clock analysis of synchronous data-flow programs. In *Proceedings of the 1995's ACM Conference on Partial Evaluation and Program Manipulation.* ACM, 1995.
7. M. Le Borgne. Systèmes dynamiques polynomiaux sur les corps finis. *Ph. D. Thesis*, Université de Rennes I, September 1991.
8. O. Maffeïs and P. Le Guernic. Distributed implementation of SIGNAL: scheduling and graph clustering. In *3rd. International Symposium on Formal techniques in Real-Time and Fault-Tolerant Systems*, p. 547–566. LNCS no. 863, Springer Verlag, 1995.
9. D. Nowak, J.-P. Talpin, T. Gautier, and P. Le Guernic. An ML-like module system for the synchronous language Signal. Submitted for publication, December 1996.

A Last Example

In this appendix, we consider a reasonably scaled SIGNAL process which makes extensive use of constants, of down-sampling and of feed-back loops using delayed signals.

```
process level (in event fill; out logical empty) =
        ( synchro (when m = 0, fill)
        | n := (10 when fill) default (m - 1)
        | m := n $ 1
        | empty := when (n = 0) default (not fill)
        ) / integer n, m init 0
```

The process level models a system similar to a water reservoir. The input event fill signals that the resource is filled. The local integer variable n measures the current level of water. At each fill event, the level is set to the maximum 10. Then, the level gradually decreases until it reaches 0. In this case, the system outputs the value true to the signal empty. Let us define f for fill and e for empty. Then, the process level has reactive type:

$$\exists n.((((\perp \ominus f) \oplus \delta n) \to n) \otimes (f \times (\perp \ominus \delta n)) \otimes ((\perp \ominus n) \oplus f) \to e$$

As a matter of comparison, the transitive closure of the graph inferred by the clock calculus of SIGNAL (figure 4) for the process level is:

$$\exists c_{1,..5}, m. \begin{pmatrix} \hat{f} = \hat{c_1} \wedge [c_2] \\ \hat{e} = (\hat{c_4} \wedge [c_5]) \vee \hat{f} \\ f \xrightarrow{\hat{f} \setminus (\hat{c_4} \wedge [c_5])} e \end{pmatrix} \; s.t. \; \hat{c_2} = \hat{c_5} = \hat{m} = (\hat{c_3} \wedge \hat{f}) \vee \hat{m}$$