An ML-Like Module System for the Synchronous Language SIGNAL

David Nowak, Jean-Pierre Talpin, Thierry Gautier and Paul Le Guernic

IRISA (INRIA-Rennes & CNRS) Campus de Beaulieu, F-35000 Rennes

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. To enable separate compilation of the functional components of reactive systems while preserving their global integrity, we introduce a module system for SIGNAL. Just as data-types describe the invariants of program modules in functional languages, temporal and data-flow invariants interface SIGNAL processes to their environment. In conventional languages, typing is the medium allowing the separate compilation of functions in a program. In SIGNAL, the notion of conditional data-flow graph can similarly be used for separately compiling reactive processes and for assembling them in complex systems. Following this principle, we present the first design and implementation of a polymorphic type system and of a module system for the synchronous language SIGNAL.

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 [2], LUSTRE [4] or ESTEREL [3], STATEMATE [5], 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 a module system for SIGNAL. Just as data-types describe the invariants of program modules in functional languages, temporal and data-flow invariants interface SIGNAL processes to their environment. In conventional languages, typing is the medium enabling the separate compilation of the functions of a program. In SIGNAL, conditional data-flow graphs can similarly be used for separately compiling reactive processes.

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.

Syntax The syntax of SIGNAL is defined in the figure 1. An expression e is either a value v (event, boolean, integer, real, etc), a signal x, the reference $x\$1$ to the previous value of a signal x, the down-sampling x when y of the signal x when y is present and true, the deterministic merge x default y of two signals x and y, or a synchronous operation $o(x_{1..n})$ (e.g. boolean, numerical operations). A process p is either an equation $x := e$, the synchronous composition of two processes, or a call $y_{1..m} := f(x_{1..n})$ to a process f with input sequence $x_{1..n}$ and output sequence $y_{1..m}$. A declaration p/x or d defines either a local signal x or a process f of inputs $x_{1..n}$, outputs $y_{1..m}$ and body p.

v	value	$d ::= \mathsf{process}\ f(?\ x_{1..n}\ !\ y_{1..m})\ p$	definition
x, y, z	signal	$e ::= v \mid x \mid x\$1\ \mathsf{init}\ v \mid x\ \mathsf{when}\ y \mid x\ \mathsf{default}\ y \mid o(x_{1..n})$	expression
f	process	$p ::= (x := e) \mid (p \parallel p') \mid y_{1..m} := f(x_{1..n}) \mid p/x \mid p\ \mathsf{where}\ d$	process

Fig. 1. Syntax of SIGNAL

Dynamic Semantics The dynamic semantics of SIGNAL, written $p \xrightarrow{m} p'$, presented in [2], 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 an instantaneous transition from a state p to a state p' and by a set m of simultaneous events (written $x(v)$ for x present with v or $x(\perp)$ for x absent).

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

$x := v$	$\xrightarrow{x(v)}$	$x := v$	$x := y$ when z	$\xrightarrow{x(\perp)\ y(\perp)}$	$x := y$ when z
$x := o(y, z)$	$\xrightarrow{x(\perp)\ y(\perp)\ z(\perp)}$	$x := o(y, z)$	$x := y$ when z	$\xrightarrow{x(v)\ y(v)\ z(t)}$	$x := y$ when z
$x := o(y, z)$	$\xrightarrow{x(o(v,v'))\ y(v)\ z(v')}$	$x := o(y, z)$	$x := y$ when z	$\xrightarrow{x(\perp)\ y(v)\ z(f)}$	$x := y$ when z
$x := y\$1$ init v	$\xrightarrow{x(\perp)\ y(\perp)}$	$x := y\$1$ init v	$x := y$ default z	$\xrightarrow{x(v)\ y(v)}$	$x := y$ default z
$x := y\$1$ init v	$\xrightarrow{x(v)\ y(v')}$	$x := y\$1$ init v'	$x := y$ default z	$\xrightarrow{x(v)\ y(\perp)\ z(v)}$	$x := y$ default z
$x := y$ when z	$\xrightarrow{x(\perp)\ z(\perp)}$	$x := y$ when z	$x := y$ default z	$\xrightarrow{x(\perp)\ y(\perp)\ z(\perp)}$	$x := y$ default z

Fig. 2. Dynamic semantics of SIGNAL

Parallel composition $p \mid p'$ synchronizes the events m and m' produced by p and p'. The relation $m \cap m'$ is defined iff. $m(x) = m'(x)$ for all x in $dom(m) \cap dom(m'))$. A synchronous operation $x := o(y, z)$ instantaneously computes the value of o by y and z and outputs it to the signal x. A delay $x := y\$1$ init v stores the value v' of y and outputs the previous value v to x. A merge $x := y$ default z outputs the value of y to x when y is present (z is not needed) and the value of z otherwise (y is absent). A sampling $x := y$ when z outputs the value of y to x when z is present and true. When the inputs of an equation are absent, a transition takes place but no value is given to its output. The dynamic semantics of a definition p where process $f(?\, x_{1..n} \,!\, x_{n+1..m})\, p'$ is equivalent to that of p where every expression $y_{n+1..m} := f(y_{1..n})$ is executed as $p'[y_{1..m}/x_{1..m}]$.

Compilation The compilation of SIGNAL requires the static resolution of boolean equations where signals are abstracted by clocks. The SIGNAL compiler ensures the respect of the synchronization constraints expressed in programs by computing temporal relations between signal clocks and analyzing the conditional data dependencies between signals. The control model of a SIGNAL program is represented by a set of temporal relations $\hat{x} = c$ between signal clocks $\hat{x}$ and expression clocks c. The clock $\hat{x}$ denotes the instants when x is present. The clock $[x]$ denotes the presence of a boolean signal x with the value true. The clock $\hat{x} \backslash \hat{y}$ 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. $x \notin fv(g) \cup bv(g)$. In most cases, references to bound signals in a graph g can be eliminated by determining its transitive closure on input/output signals[1].

$$c ::= \hat{x} \mid \hat{x} \backslash \hat{y} \mid [x] \mid c \wedge c \mid c \vee c \qquad g ::= \emptyset \mid g \cup (\hat{x} = c) \mid g \cup (x \xrightarrow{c} y) \mid \exists x.g \mid g \cup g'$$

Fig. 3. Conditional data-flow graphs g

Clock Calculus The clock calculus of SIGNAL, defined by the proof system $G \vdash p \Rightarrow g$ of the figure 4, determines the conditional data-flow graph g of programs[2].

$$G \vdash x := y\$1 \Rightarrow (\hat{x} = \hat{y}) \qquad G \vdash x := y \text{ when } z \Rightarrow (y \xrightarrow{[z]} x, \hat{x} = \hat{y} \wedge [z])$$

$$G \vdash x := o(y_{1..n}) \Rightarrow \left(\begin{array}{l} y_i \xrightarrow{\hat{y}_i} x \\ \hat{x} = \hat{y}_i \end{array} \right)_{i=1..n} \qquad G \vdash x := y \text{ default } z \Rightarrow \left(\begin{array}{l} y \xrightarrow{\hat{y}} x, z \xrightarrow{\hat{z} \backslash \hat{y}} x \\ \hat{x} = \hat{y} \vee \hat{z} \end{array} \right)$$

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

$$\frac{f : \forall x_{1..m}.g \in G}{G \vdash y_{n+1..m} := f(y_{1..n}) \Rightarrow g[y_{1..m}/x_{1..m}]} \qquad \frac{G \vdash p' \Rightarrow g' \quad G \cup \{f : \forall x_{1..m}.g'\} \vdash p \Rightarrow g}{G \vdash p \text{ where process } f(?\, x_{1..n} \,!\, x_{n+1..m})\, p' \Rightarrow g}$$

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

[1] $g|_x = g \backslash g_x \backslash g^x \cup \{(y \xrightarrow{c \wedge c'} z) \mid (y \xrightarrow{c} x, x \xrightarrow{c'} z) \in g_x \times g^x\}$ is the closure of g w.r.t. x where $g^x = \{x \xrightarrow{c} y \in g\}$ and $g_x = \{y \xrightarrow{c} x \in g\}$

[2] The graph environment G associates process names f to process specifications $\forall x_{1..n}.g$

Using the graph g produced by the clock calculus, the SIGNAL compiler generates an optimal compile-time scheduling of the actions specified in the source program by hierarchizing temporal relations in g and by ruling the execution of the program using its master clock [1]. 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. $x \xrightarrow{c} x$). This allows proving the absence of dead-locks in the program.

Example 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). To compile it, the SIGNAL compiler has the choice between scheduling either if present(a) x:=a; if present(b) y:=b or if present(b) y:=b; if present(a) x:=a. However, the appropriate choice depends on the way the process is invoked. For instance, if we write (u,v) := copy(w,u), only the first scheduling (i.e. if present(w) {u:=w; v:=u}) is correct (the second dead-locks). 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 separate compilation.

3 Polymorphic Type Inference in SIGNAL

In this section, we propose a polymorphic type system for SIGNAL. The type system complements the clock calculus of SIGNAL with the static determination of signal types. Just as the clock calculus allows the detection of non-reactive or dead-lockingSIGNAL programs, the type inference system allows the detection of erroneous access to data-structures. For instance, consider the definition of the generic identity function id applied to either integers or booleans below. The SIGNAL compiler [6] handles this specification by textually substituting the definition of the generic process id everywhere the process is used. This results in constructing a program in elementary syntax: "a:=true ‖ b:=1". Type checking is then performed on the elementary syntax. However, if the process id is separately compiled, this translation is not possible. The introduction of type polymorphism is the solution to this issue. Just as the conditional data-flow graph $x \xrightarrow{\hat{x}} y \ s.t. \ \hat{x} = \hat{y}$ generically represents the behavior of id, the polymorphic type $\forall\alpha.\alpha \rightarrow \alpha$ characterizes the type of all its possible instances.

```
(a := id(true) | b := id(1)) where process id (? x ! y) (y := x)
```

Type System The type system of SIGNAL has a rich structure to handle the variety of data structures and formats of real-time systems. Some predefined data-types are given in the figure 5 in order to address an important issue: subtyping. The smallest signal type t is the event and denotes a boolean which is always true when present. A type variable α denotes a generic signal type.

The types t of the signals manipulated by a process are subject to simple[3] subtyping constraints denoted by a set C. The subtyping relation, written "$t \leq t'$", reads "the signal type t is a subtype of t'" and is defined as follows.

$$t ::= \mathsf{event} \mid \mathsf{boolean} \mid \mathsf{short} \mid \mathsf{integer} \mid \mathsf{long} \mid \mathsf{real} \mid \mathsf{double} \mid \alpha \qquad \text{signal types}$$

$$\tau ::= t_{1..n} \to t'_{1..m} \text{ process types} \qquad A ::= \emptyset \mid A \cup \{x:t\} \mid A \cup \{f:\sigma\} \text{ assumptions}$$
$$\sigma ::= \tau / C \mid \forall \alpha.\sigma \quad \text{generic types} \qquad C ::= \emptyset \mid C \cup \{t \leq t\} \qquad \text{constraints}$$

Fig. 5. Type system

$$C \supset \mathsf{short} \leq \mathsf{integer} \qquad C \supset \mathsf{integer} \leq \mathsf{long} \qquad C \supset \mathsf{real} \leq \mathsf{double} \qquad C \supset t \leq t$$

$$C \cup \{t \leq t'\} \supset t \leq t' \qquad \frac{C \supset t \leq t' \quad C \supset t' \leq t''}{C \supset t \leq t''} \qquad \frac{C \supset t \leq t' \quad C \supset t' \leq t}{C \supset t = t'}$$

Fig. 6. Subtyping relation

Type Polymorphism The type τ of a process is a map $t_{1..n} \to t'_{1..m}$ from inputs types $t_{1..n}$ to output types $t'_{1..m}$. A polymorphic process type σ is a process type τ universally quantified over the set of its generic signal type variables α (eventually constrained by a set C of subtyping relations). Assumptions A in the type inference system declare signal types "$x:t$" and process types "$f:\sigma$". The generalization and the instantiation of process types is defined in the figure 7. The function fv gives the set of free signal variables α of a term (a type t, a polymorphic type σ or an environment A). We write $\overline{C}$ for the transitive closure of C for the relation $\leq$ (i.e. $t \leq t'' \in \overline{C}$ for all $(t \leq t'), (t' \leq t'') \in C$). We write $\overline{C}_{\alpha_{1..n}}$ for the restriction of $\overline{C}$ to all constraints referencing $\alpha_{1..n}$.

$$gen(A, C, \tau) = \forall \alpha_{1..n}.(\tau / \overline{C}_{\alpha_{1..n}}) \text{ iff. } \alpha_{1..n} = (fv(\tau) \cup fv(C)) \backslash fv(A)$$
$$\tau / C \preceq \forall \alpha_{1..n}.(\tau' / C') \text{ iff. } \tau = \tau'[t_{1..n}/\alpha_{1..n}] \text{ and } C \supset C'[t_{1..n}/\alpha_{1..n}]$$

Fig. 7. Type generalization and instantiation

Type Inference System The type inference system of SIGNAL is defined in the figure 8 by the sequent $A, C \vdash p$ which reads "p is well-typed with the assumptions A and constraints C". The auxiliary sequent $A, C \vdash e:t$ checks that the expression e has signal type t under the assumptions A and constraints C.

$$\frac{A(x) = t}{A, C \vdash x:t} \qquad \frac{\mathcal{R}(v) = t \quad A, C \vdash x:t}{A, C \vdash x\$1 \text{ init } v:t} \qquad \frac{A, C \vdash e:t \quad C \supset t \leq t'}{A, C \vdash e:t'} \qquad \frac{A, C \vdash x:t \quad A, C \vdash y:t}{A, C \vdash x \text{ default } y:t}$$

$$\frac{\mathcal{R}(v) = t}{A, C \vdash v:t} \qquad \frac{A, C \vdash x:t \quad A, C \vdash y:\mathsf{boolean}}{A, C \vdash x \text{ when } y:t} \qquad \frac{A, C \vdash x:t \quad A, C \vdash e:t}{A, C \vdash x := e} \qquad \frac{A \cup \{x:t\}, C \vdash p}{A, C \vdash p/x}$$

$$\frac{A, C \vdash p \quad A, C \vdash p'}{A, C \vdash p \mid p'} \qquad \frac{(A, C \vdash x_i:t_i)_{i=1..m} \quad t_{1..n} \rightarrowtail t_{n+1..m} / C \preceq A(f)}{A, C \vdash x_{n+1..m} := f(x_{1..n})}$$

$$\frac{A \cup \{x_i:t_i, i = 1..m\}, C \vdash p' \quad A \cup \{f:gen(A, C, t_{1..n} \to t_{n+1..m})\}, C \vdash p}{A, C \vdash p \text{ where process } f(?\ x_{1..n}\ !\ x_{n+1..m})\ p'}$$

Fig. 8. Type inference system

[3] Note that signal types, and only signal types t, are referenced in subtyping constraint sets C. Hence, each constraint references either ground types (e.g. event) or type variables α.

The function $\mathcal{R}$ relates each constant to its minimal predefined type. The type inference system proceeds by structurally scanning expressions e and processes p to determine output signal types t and subtyping constraints C. All rules but the rule of subtyping apply on a specific syntactic category. Every expression rule $A, C \vdash e : t$ explicits a type containment constraint. For instance, the rule for a merge statement "x default y" says that the type of both x and y must match t for the typing of the expression to be preserved. The rule of subtyping is however applicable either to x or to y in order to eventually coerce types to a common upper bound. The rules for processes p, written $A, C \vdash p$ differ in that they just check that the process p agrees with a set of assumptions A and a set of constraints C i.e. that A and C are sufficient to give a type to every expression in p. Process definitions and applications make use of type polymorphism in order to represent and use all the possible types of a process. A type reconstruction algorithm can easily be derived from the inference system of the figure 8 by composing every expression rule $A, C \vdash e : t$ with the subtyping rule and by introducing a fresh signal type α wherever a type t is introduced or referenced in the expression rules. In addition, some simplifications rules for constraints sets can be implemented by removing trivial constraints (e.g. constraints between ground types or constraints on unreferenced type variables) and by substituting constraints between type variables α and minima (e.g. $\alpha \leq \mathsf{event}$) or maxima (e.g. $\mathsf{boolean} \leq \alpha$). The definition and proof of a type reconstruction algorithm is presented in [7].

Correctness of the Inference System The correctness of the type inference system with respect to the dynamic semantics is proven by showing that typing is an invariant of process execution (theorem 1). We say that an event map m is consistent with A and C, written $A, C \vdash m$, iff. $A, C \vdash m(x) : A(x)$ for all $x \in dom(m)$. We write $out(p)$ (resp. $in(p) = fv(p) \setminus out(p)$) for the input/output signals of p.

Theorem 1. *If* $A, C \vdash p$, $A, C \vdash m_{in(p)}$ *and* $p \xrightarrow{m} p'$ *then* $A, C \vdash p'$ *and* $A, C \vdash m_{out(p)}$.

4 A Module System for SIGNAL

In the previous sections, we did present the clock calculus and the polymorphic type inference system of SIGNAL. Just as data-types describe the invariants of program modules in functional languages, temporal and data-flow invariants interface SIGNAL processes to their environment. In conventional languages, typing is the medium enabling the separate compilation of the functions of a program. In SIGNAL, conditional data-flow graphs can similarly be used as a medium for separately compiling reactive processes. The module system presented in this section put this principles to work by defining a notion of specification and of implementation for all components of a program. In addition, modules can be parameterized by other modules, in a way which was first introduced in STANDARD ML [8, 9] and CAML [10]. The integration of the principles of type polymorphism and modules to SIGNAL form a system which considerably augments the expressive power of SIGNAL and generalizes its underlying programming methodology. We present a generalized notion of signature subtyping which allows characterizing the refinement of modular components of reactive systems. This notion of refinement makes use of the notion of binary decision diagrams in order to prove that the invariants expressed in the signature of a module are more general than the invariants inferred from its implementation.

Module Language The syntax of the language which implements the module system of SIGNAL is defined in the figure 9. It consists of signatures S, modules M and functors F. The figure 9 defines the syntax of programs P as a sequence of signatures S, functors F and module declarations M. A signature s gives the specification or interface of a module in terms of type declaration and process specification. A process specification "process $f(?\,x_1 : t_1 .. !\,..x_n : t_n)/C$ spec g" consists of the declaration of its input/output signal types $t_{1..n}$ and of its temporal and data-flow invariants modeled by the conditional data-flow graph g. A functor declaration is a module object parameterized over a sequence of modules. A module declaration gives the implementation of a program component.

$$
\begin{array}{lll}
P ::= & \mathsf{signature}\ S = s\ \mathsf{end} & s ::= \quad s; s' \\
& \mid \mathsf{functor}\ F(M_{1..n} : S_{1..n}) = m\ \mathsf{end} & \quad \mid \mathsf{type}\ T \\
& \mid \mathsf{module}\ M = F(M_{1..n}) & \quad \mid \mathsf{type}\ T = t \\
& \mid \mathsf{module}\ M = m\ \mathsf{end} & \quad \mid \mathsf{process}\ f(?\,x_1 : t_1 .. !\,..x_n : t_n)/C\ \mathsf{spec}\ g \\
& \mid P; P' & m ::= \ \mathsf{type}\ T = t \mid d \mid m; m'
\end{array}
$$

Fig. 9. Syntax of the module system

Subtyping of Signatures The key principle of a module system is that the interface or signature of a module should contain enough information to allow separate verification and implementation of other modules using it. The notion of signature matching and subtyping is central for checking the correctness of this information flow. In conventional languages, such as STANDARD ML [8, 9], this amounts to defining a relation $s \leq s'$ between signatures which specifies the agreement protocol between the interface of a module and its implementation.

$$
A \vdash \mathsf{type}\ T \leq \mathsf{type}\ T \qquad A \vdash \mathsf{type}\ T = t \leq \mathsf{type}\ T \qquad \frac{A \vdash t \simeq t'}{A \vdash \mathsf{type}\ T = t \leq \mathsf{type}\ T = t'}
$$

$$
\frac{A(T) = t}{A \vdash T \simeq t} \qquad \frac{t'_{1..n} \to t'_{n+1..n'}/C' \preceq gen(A, C, t_{1..n} \to t_{n+1..n'}) \quad g \leq g'[x_{1..n'}/y_{1..n'}]}{\begin{array}{l} A \vdash \mathsf{process}\, f(\, x_{1..n} : t_{1..n}\, !\, x_{n+1..n'} : t_{n+1..n'})/C\ \mathsf{spec}\, g \\ \leq \mathsf{process}\, f(?\, y_{1..n} : t'_{1..n}\, !\, y_{n+1..n'} : t'_{n+1..n'})/C'\ \mathsf{spec}\, g' \end{array}}
$$

Fig. 10. Subtyping of signatures

This relation is defined in the figure 10. Unlike for conventional modules systems, it does not reduce to checking the containment of type definitions and of process types declarations. Let us consider, as in the above figure, a process signature of declared data-flow graph g' and d be its implementation of data-flow graph g. Checking that the temporal and data-flow information propagated in the signature (i.e. g') contains that inferred in the implementation of the process (i.e. g) amounts to verifying the relation written $g \leq g'$.

Definition 2. For all g, $\overrightarrow{g} = [(x \xrightarrow{c} y) \in g]$ and $\hat{g} = [(\hat{x} = c) \in g]$. For all g and g', $g \leq g'$ iff. $\overrightarrow{g} \subseteq \overrightarrow{g'}$ and $\hat{g'} \Rightarrow \hat{g}$.

The relation $g \leq g'$ guaranties the safety of the interface of a module with respect to its implementation. The interface must declare any interaction performed in the implementation of a module in order to verify the safety of the use of the module.

The refinement of $\overrightarrow{g}$ by a bigger $\overrightarrow{g'}$ ensures that whenever $\overrightarrow{g'}$ is safe (it does not contain cyclic data dependencies $x \xrightarrow{c} x$ with non-trivial clock c) then so is $\overrightarrow{g}$. The refinement of $\hat{g}$ by $\hat{g'}$ means that every clock equation in $\hat{g}$ can be deducted from $\hat{g'}$. Hence, a trivial cycle $x \xrightarrow{c} x$ in g (i.e. s.t. $\neg c$) cannot be non-trivial in g'. Furthermore, a clock equation in g' cannot induce a causal dependency of the form $[y] = c$ in g.

Refinement Checking Checking satisfaction of the relation $g \leq g'$ amounts to solving the problem $\hat{g'} \Rightarrow \hat{g}$. As $\hat{g'}$ and $\hat{g}$ are systems of boolean equations, this problem can be expressed on $Z/2Z$ [1] as $\hat{g} : (P_i(\overrightarrow{x}) = 0)_{i \in I}$ and $\hat{g'} : (P'_j(\overrightarrow{x}) = 0)_{j \in J}$ where the formula P_i and P'_j (of index I and J) are polynomials on $Z/2Z$ and where $\overrightarrow{x}$ is the vector of signals defined in the system of equations. Now, let us define the boolean operator $\oplus$ as $a \oplus b = a + b + a.b$. A property of $\oplus$ is that $P_1(\overrightarrow{x}) = 0$ and $P_2(\overrightarrow{x}) = 0$ iff. $(P_1 \oplus P_2)(\overrightarrow{x}) = 0$. Using $\oplus$, the system of equations reduces to $\hat{g} : P(\overrightarrow{x}) = 0$ and $\hat{g'} : P'(\overrightarrow{x}) = 0$ where $P = \oplus_{i \in I} P_i$ et $P' = \oplus_{j \in J} P_j$. Let V and V' be the solutions of P and P', $V \subseteq V'$ iff. $\forall \overrightarrow{x}\,.(1 - P(\overrightarrow{x})).P'(\overrightarrow{x}) = 0$, which amounts to proving a tautology.

Signature Checking Having formally defined a notion of signature matching in our module system allows us now to define signature checking by the sequent $A \vdash P : A'$ in the figure 11.

$$A \vdash \mathsf{signature}\ S = s\ \mathsf{end} : A \cup \{S : s\} \qquad A \cup \{M : s\} \vdash M : s \qquad A \cup \{S : s\} \vdash S : s$$

$$\frac{A(F) = (M'_{1..n} : s_{1..n}) \to s \quad (A \vdash M_i : s_i)_{i=1..n}}{A \vdash \mathsf{module}\ M = F(M_{1..n}) : A \cup \{M : s[M'_{1..n}/M_{1..n}]\}} \qquad \frac{A \vdash P : A' \quad A' \vdash P' : A''}{A \vdash P; P' : A''}$$

$$\frac{A \vdash m : s}{A \vdash \mathsf{module}\ M = m\ \mathsf{end} : A \cup \{M : s\}} \qquad \frac{A \cup \{T : t\} \vdash m : s}{A \vdash \mathsf{type}\ T = t; m : \mathsf{type}\ T = t; s}$$

$$\frac{(A \vdash S_i : s_i)_{i=1..n} \quad A \cup \{M_{1..n} : s_{1..n}\} \vdash m : s}{A \vdash \mathsf{functor}\ F(M_{1..n} : S_{1..n}) = m\ \mathsf{end} : A \cup \{F : (M_{1..n} : s_{1..n}) \to s\}} \qquad \frac{A \vdash m : s' \quad A \vdash s' \leq s}{A \vdash m : s}$$

$$\frac{\begin{array}{c} A \cup \{x_i : t_i\}_{i=1..n'}, C, G \vdash p \qquad A, G \vdash p' \Rightarrow g \\ A \cup \{f : gen(A, C, t_{1..n} \to t_{n+1..n'})\}, G \cup \{f : \forall x_{1..n'}.g\} \vdash m : s \end{array}}{A \vdash \mathsf{process}\ f(?\,x_{1..n}\,!\,x_{n+1..n'})\ p; m : \mathsf{process}\ f(?\,x_{1..n} : t_{1..n}\,!\,x_{n+1..n'} : t_{n+1..n'})/C\ \mathsf{spec}\ g; s}$$

Fig. 11. Signature checking

$$\frac{A \vdash M : s; \mathsf{type}\ T = t; s'}{A \vdash M.T \simeq t[M.T'/T']_{\mathsf{type}\ T' = t' \in s}} \qquad \frac{\mathsf{process}\ f(?\,x_1 : t_1 .. !\,..x'_n : t'_n)/C\ \mathsf{spec}\ g \in A(M)}{A, G \vdash y_{n+1..n'} := M.f(y_{1..n}) \Rightarrow g[y_{1..n'}/x_{1..n'}]}$$

$$\frac{A, C \vdash e : t \quad A \vdash t \simeq t'}{A, C \vdash e : t'} \qquad \frac{\begin{array}{c} \mathsf{process}\ f(?\,x_{1..n} : t_{1..n}\,!\,x_{n+1..n'} : t_{n+1..n'})/C'\ \mathsf{spec}\ g \in A(M) \\ A \cup \{M.f : gen(A, C \cup C', t_{1..n} \to t_{n+1..n'})\}, C \vdash y_{n+1..n'} := M.f(y_{1..n}) \end{array}}{A, C \vdash y_{n+1..n'} := M.f(y_{1..n})}$$

Fig. 12. Modular clock and type inference

The principle of the signature checker is to verify the correctness of a signature declaration w.r.t. the information inferred from its implementation, and then checking that it is correctly used elsewhere. The module checker sequentially analyses the signatures and modules declared in a program P, verifies that declarations in A match uses in P and then returns a set of assumptions A' augmented with the declarations made in the processed component P. The definition of the module checker centralizes the verification of the program by using the type inference sys-

tem $A, C \vdash p$ to check the implementation of programs, and of the clock calculus $A, G \vdash p \Rightarrow g$ in order to verify their safety. In order to be usable in the module checker, the type and clock inference systems (figures 4 and 8) need to understand type declarations and references to processes made in separate modules (figure 12).

Correctness of the Module System In a conventional module system, the type interface of a program module should contains enough information to allow its separate verification and compilation. This requirement is usually implemented using a notion of signature matching, which checks the correctness of the type-information flow. Conventionally, implementing signature matching amounts to defining a relation $s \leq s'$ between module signatures. In our system, such a relation is extended to checking the containment of temporal and data-flow information using the relation $g \leq g'$. As a result, previous formal propositions for establishing the correctness of module systems [9, 10] hold. The correctness of our module system could be stated by showing that the decomposition of a system of modules P in a context A using an appropriate sequent translation $A \vdash P \Rightarrow A', C \vdash p$ produces the appropriate context (A', C) to check the corresponding program p well-typed.

5 Implementation

Since its first implementation, more than a decade ago, the SIGNAL compiler has been in constant evolution to integrate new principles of programming, compilation and verification. The most recent version of the compiler, SIGNAL V4 [6], implements a common format called DC+ which is the result of the SYNCHRON initiative. The type inferencer described in this paper has been integrated in SIGNAL V4. Detailed proofs and examples are available in [7]. The module system is currently implemented as a separate "*module processor*", to maximize the flexibility of its use. An important point on the compilation of SIGNAL is that, although it is best to perform type inference at an early stage of the compilation process, type information in the SIGNAL compiler is not needed until the very last step of the compilation process: code generation (usually C or VHDL). Our module processor has been implemented in such a way to allow the separate verification of program modules and functors "$M \Rightarrow g$". From the implementation of a functor F or module M, the module checker can produce a specification g which can then be checked by the SIGNAL compiler for safety. The separate compilation of a hierarchy of modules "$P \Rightarrow p$" is also supported. Since code generation is the very last step of the compilation for SIGNAL programs where type information is needed, all earlier verification and optimizations can be performed on the intermediate representation of a SIGNAL program while preserving the information declared in its signature. Experiments show that extending our module system with a notion of higher-order modules (i.e. by parameterizing functors with functors as in [10], for instance), would not be technically a problem. However, from a practical point of view, it would incur the introduction of a notion of *contravariant refinement*, which would certainly be very difficult to understand and to use. From a methodological point of view, such a feature would also not provide any significant improvement over the present system. Another direction for improvements in our module system is the representation of temporal and data-flow invariants of programs using conditional data-flow graph. We have seen that the interaction on

local signals is represented using existential quantification. A general method for eliminating such quantifications would either be to introduce clock inequations or use reactive types, as advocated in [12].

6 Conclusion

We have presented the design and implementation of a polymorphic type inferencer and of a module system for the synchronous language SIGNAL. This system is inspired from principles introduced in previous implementations of functional languages such as STANDARD ML [8, 9] or CAML [10]. Our main contribution is to show that, just as data-types describe the invariants of program modules in functional languages, temporal and data-flow invariants interface SIGNAL processes to their environment: the notion of signature matching employed in conventional module systems naturally extends to a notion of signature refinement in synchronous languages.

References

1. 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.
2. A. Benveniste, P. Le Guernic and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. In *Science of Computer Programming*, v. 16, p. 103–149, 1991.
3. G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. In *Science of Computer Programming*, v. 19, p. 87–152, 1992.
4. N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud. The synchronous data-flow programming language Lustre. In *Proceedings of the IEEE*, v. 79(9). IEEE Press, 1991.
5. D. Harel and A. Naamad. *The* STATEMATE *semantics of* STATECHARTS. I-Logix, 1995.
6. Thierry Gautier, Paul Le Guernic, and François Dupont. SIGNAL v4 : manuel de référence. Technical report n. 832. IRISA, 1994.
7. D. Nowak, Talpin, J.-P., and Gautier, T. Un système de modules avancé pour SIGNAL. Rapport de recherche n. ??, 1997. (To appear, available from nowak@irisa.fr).
8. R. Milner, M. Tofte, R. Harper. *The definition of* STANDARD ML. MIT Press, 1990.
9. M. Tofte. Principal signatures for higher-order program modules. In *Proceedings of the 1992's ACM Symposium Principles of Programming Languages*, p. 189-199. ACM, 1992.
10. X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 1995's ACM Symposium Principles of Programming Languages*. ACM, 1995.
11. O. Maffeïs and P. Le Guernic. Distributed implementation of SIGNAL: scheduling and graph clustering. In *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Lecture Notes in Computer Science n. 863. Springer Verlag, 1994.
12. J.-P. Talpin. Reactive Types. In *Conference on the Theory and Practice of Software Development (TAPSOFT'97)*. Lecture Notes in Computer Science. Springer Verlag, 1997.