Modelling SIGNAL in Interaction Categories

Simon Gay and Rajagopal Nagarajan
{sjg3,rn4}@doc.ic.ac.uk
Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, United Kingdom

Abstract

Abramsky has recently proposed Interaction Categories as a new paradigm for the semantics of sequential and parallel computation. Working with the category **SProc** of synchronous processes, which is a key example of an Interaction Category, we study synchronous dataflow as part of a programme of gaining experience in the use of Interaction Categories. After making some general points about representing dataflow in **SProc**, we present a detailed model of the synchronous dataflow language SIGNAL. We demonstrate that dataflow is a model of concurrency which can easily be treated in a typed framework, and that the structure of Interaction Categories is appropriate for describing real concurrent languages.

1 Motivation

Abramsky [1] has recently proposed a new paradigm for the semantics of sequential and concurrent computation: Interaction Categories. This term encompasses certain known categories (the category of concrete data structures and sequential algorithms [5], categories of games [2]) as well as a new category **SProc**, with which we will be working in this paper. The distinguishing feature of Interaction Categories is that composition in them is a dynamic process of *interaction*, rather than the static one of function composition found in the familiar categories of traditional mathematics. **SProc** can be read either as "Synchronous Processes" or "Specifications and Processes"; the present work leans towards the first reading. In particular, we are using **SProc** to model SIGNAL [8] - one of a family of synchronous programming languages. Other members of the family include LUSTRE [9], SILAGE [10] and ESTEREL [4]. Synchrony is easier to handle initially; our aim is to deal with asynchrony later. Independently of this, dataflow is a model of concurrency which is well-suited to a typed framework. Thus we choose the synchronous dataflow language SIGNAL as our starting point. LUSTRE has also been modelled; this work is described elsewhere [7]. The purpose of this effort is two-fold: to see how existing programming paradigms are supported in the new framework, and to obtain feedback about how appropriate the categorical structures are.

2 The Interaction Category SProc

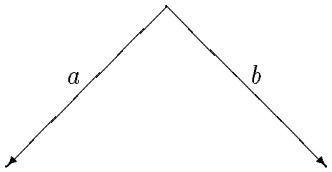
We will begin with a brief review of the definition of **SProc**, highlighting those features which are relevant to modelling dataflow. The basic picture is

Objects	Specifications
Morphisms	Processes
Composition	Synchronous composition + restriction.

This category is a model of 2nd-order Linear Logic, and supports polymorphism. It also includes a hierarchy of delay operators (as *monads*) and hence allows asynchrony to be built on top of synchrony *à la* Milner [12]. In addition, specification and verification can be smoothly incorporated via the notion of *specification strutures*, although we are not making use of this as yet. The following definitions relating to **SProc** are based on Abramsky's work.

2.1 Basic Definitions

The objects of **SProc** are pairs $A = (\Sigma_A, S_A)$ where Σ_A is an alphabet of actions (labels) and $S_A \subseteq^{\text{nepref}} \Sigma_A^*$, a non-empty prefix-closed subset of the set of finite sequences over Σ_A, is a safety specification. A *process* of type A, written $p \models A$, is a synchronisation tree with labels from Σ_A, such that $\mathsf{traces}(p) \subseteq S_A$. Following Aczel [3], we will make use of a representation of synchronisation trees as (non-well-founded) sets, in which a process p with transitions $p \xrightarrow{a} q$, $p \xrightarrow{b} r$ becomes $\{(a,q),(b,r)\}$; the synchronisation tree looks like



at the top. In order to define the morphisms of **SProc**, we first define a $*$-autonomous structure. Given A and B, the object $A \otimes B$ has

$$\begin{aligned} \Sigma_{A\otimes B} &= \Sigma_A \times \Sigma_B \\ S_{A\otimes B} &= \{\sigma \in \Sigma^*_{A\otimes B} \mid \mathsf{fst}^*(\sigma) \in S_A \wedge \mathsf{snd}^*(\sigma) \in S_B\}. \end{aligned}$$

The duality is trivial on objects: $A^\perp = A$ (this comes from the choice of linear maps). Hence all the multiplicative connectives are the same: $A \parr B = A \multimap B = A \otimes B$. Finally, a morphism $p : A \to B$ is a process p such that $p \models A \multimap B$. Since $\otimes$ is self-dual, we have not only a $*$-autonomous category but a *compact-closed* category.

Composition is defined in line with the slogan "relational composition extended in time". If $p : A \to B$ and $q : B \to C$, so that $p \models A \multimap B$ and $q \models B \multimap C$, then the composite $p;q : A \to C$ can be defined by labelled transitions:

$$\frac{p \xrightarrow{(a,b)} p' \qquad q \xrightarrow{(b,c)} q'}{p;q \xrightarrow{(a,c)} p';q'}$$

in which matching of actions takes place in the common type B (as in relational composition), at each time step. This is the "interaction" of Interaction Categories.

The identity morphisms are synchronous buffers: whatever is received by $\mathsf{id}_A : A \to A$ in the left copy of A is instantaneously transmitted to the right copy. We define the identity proto-morphism $\mathsf{id} = \{((a,a),\mathsf{id}) \mid a \in \Sigma_A\}$ and then obtain id_A by restricting to synchronisation trees whose traces are all in $S_{A \multimap A}$: $\mathsf{id}_A = \mathsf{id}\restriction_{S_{A \multimap A}}$.

We extend $\otimes$ and $(\cdot)^\perp$ to functors by defining their action on morphisms as follows. If $p : A \to C$ and $q : B \to D$ then $p \otimes q : A \otimes B \to C \otimes D$ is defined by

$$\frac{p \xrightarrow{(a,c)} p' \qquad q \xrightarrow{(b,d)} q'}{p \otimes q \xrightarrow{((a,b),(c,d))} p' \otimes q'}$$

and $p^\perp : C \to A$ by

$$\frac{p \xrightarrow{(a,c)} p'}{p^\perp \xrightarrow{(c,a)} p'^\perp.}$$

The tensor unit I is defined by $\Sigma_I = \{*\}$, $S_I = \{*^n \mid n < \omega\}$. We also have $\perp = I$. The correct notion of "point" in a $*$-autonomous category is a morphism from I, and indeed we can identify a process of type A with a morphism $p : I \to A$. This will be important for our later work, as will the rest of the $*$-autonomous structure (i.e. closure), which we now define.

If $p : A \otimes B \to C$ then $\Lambda(p) : A \to B \multimap C$ is defined by

$$\frac{p \xrightarrow{((a,b),c)} q}{\Lambda(p) \xrightarrow{(a,(b,c))} \Lambda(q).}$$

The evaluation morphism

$$\mathsf{Ap}_{A,B} : (A \multimap B) \otimes A \to B$$

is defined via a proto-morphism in the same way as the identities:

$$\begin{aligned} \mathsf{Ap} &= \{((((a,b),a),b),\mathsf{Ap}) \mid a \in \Sigma_A, b \in \Sigma_B\} \\ \mathsf{Ap}_{A,B} &= \mathsf{Ap}\restriction_{S_{((A \multimap B) \otimes A) \multimap B}}. \end{aligned}$$

We see that the "logical" morphisms giving the closed structure are essentially formed from identities.

We will make essential use of the compact closed structure of **SProc** later, in the following way. If we have a process p of type $A \otimes B$, $p \models A \otimes B$, then we can regard it as a morphism $p : I \to A \otimes B$. Since $(\cdot)^\perp$ is trivial, this is also $p : I \to A^\perp \otimes B$, i.e. $p : I \to A \multimap B$. Hence $p = \Lambda(q)$ where $q : A \to B$. From the definition of $\Lambda(\cdot)$ we can see that q is morally the same process as p; we will abuse notation in the rest of the paper by calling them both p, freely using compact closedness to move types back and forth across arrows.

2.2 Delay

A further part of the structure of **SProc** which we will need when modelling SIGNAL is the delay monads. So far, in the synchronous world to which **SProc** corresponds, any process has to do a genuine action at every time step. We now allow for processes which perform "dummy" or "idle" actions some of the time. There are two delay monads, corresponding to the delay operators in CCS - δ which allows for delay before the first action, and Δ which allows for delay anywhere except before the first action.

Given an object A, δA is defined by

$$\Sigma_{\delta A} = \mathbf{1} + \Sigma_A$$

where for notational convenience we take $\mathbf{1} = \{*\}$ and assume $* \notin \Sigma_A$, and

$$S_{\delta A} = \{\varepsilon\} \cup \{*^n \sigma \mid (n < \omega) \wedge (\sigma \in S_A)\}.$$

If $p : A \rightarrow B$ then

$$\delta p = \{((*, *), \delta p)\} \cup p.$$

Thus δp delays for a while and then behaves as p. ΔA is defined by

$$\Sigma_{\Delta A} = \mathbf{1} + \Sigma_A$$

and

$$S_{\Delta A} = \{\varepsilon\} \cup \{a_1 *^{n_1} a_2 *^{n_2} a_3 \ldots \mid (n_i < \omega) \wedge (a_1 a_2 a_3 \ldots \in S_A)\}.$$

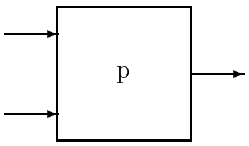
Δp is like δp with $*$ appearing anywhere except before the first action of p.

It turns out that both of these functors have monad structures. What we will actually be interested in later on is the combined delay functor $\delta\Delta$, which is also a monad by virtue of the fact that there is a distributive law, i.e. a natural transformation $\Delta\delta \stackrel{\cdot}{\rightarrow} \delta\Delta$.

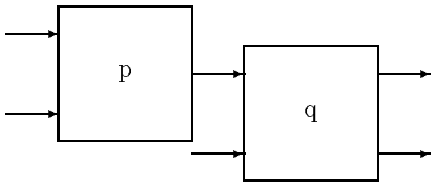
3 Dataflow in SProc

Having discussed the relevant properties of **SProc**, we can now talk about modelling dataflow in general terms, before moving on to a specific language. **SProc** gives a framework for plugging processes together on typed interfaces, which is what we want as a basis for a typed theory of concurrency. General process calculi such as CCS do not immediately fit into this framework, because of the fact that their parallel composition does not actually corresond to plugging processes together but rather to placing them side-by-side in a way which means that they *may* communicate. What we need to do is work with *restricted* composition, in which two processes in parallel which can communicate with each other are prevented from communicating with anything else. Restricted composition does correspond to connecting processes rigidly together, and this is precisely what happens in a dataflow language as networks are built out of basic nodes. Thus dataflow is a model of concurrency which can very naturally be modelled in **SProc**.

We now describe, in a very general way, how the structure of a compact closed category allows us to construct dataflow networks in a typed framework. Suppose we are working with networks in which all values come from some type A, and we use an object A in our category to model this type. A node



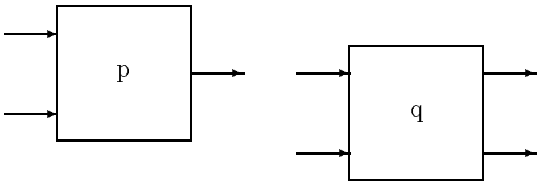
is modelled by a process (morphism) $p : A \otimes A \rightarrow A$. Now suppose we have another node q and we wish to connect the two together to form a simple network.



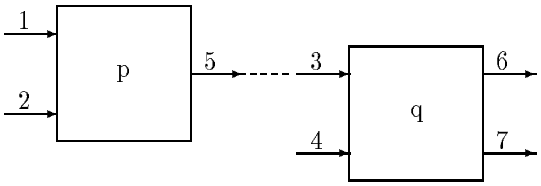
The node q is modelled by a morphism $q : A \otimes A \rightarrow A \otimes A$, and to model the network we form

$$(p \otimes \mathsf{id}_A); q : A \otimes A \otimes A \rightarrow A \otimes A.$$

Alternatively, we can take the view that to construct our network we first place p and q next to each other without communication to form a network with four inputs and three outputs



and then add an edge connecting one of the outputs to one of the inputs.



To model this we first form

$$p \otimes q : A_1 \otimes A_2 \otimes A_3 \otimes A_4 \rightarrow A_5 \otimes A_6 \otimes A_7,$$

where for clarity we have numbered the occurrences of A. Call this process r. By compact closure we can view this as a morphism

$$r : A_1 \otimes A_2 \otimes A_4 \rightarrow A_3^{\perp} \otimes A_5 \otimes A_6 \otimes A_7,$$

where we have retained the $^{\perp}$ on A_3 to indicate that what was previously seen as an input is now being viewed as an output. To get the effect of adding

the connection between output 5 and input 3, we apply $\mathsf{Ap}_{A,\perp}$ to $A_3^{\perp}$ and A_5. Recall the general definition of $\mathsf{Ap}_{A,B}$; in the case $B = \perp$ we have

$$\mathsf{Ap}_{A,\perp} : (A \multimap \perp) \otimes A \to \perp,$$

i.e.

$$\mathsf{Ap}_{A,\perp} : A^{\perp} \otimes A \to I$$

since $\perp = I$. This morphism forces the same actions to occur in $A^{\perp}$ and A, which is what we need to say that two ports are connected. This gives

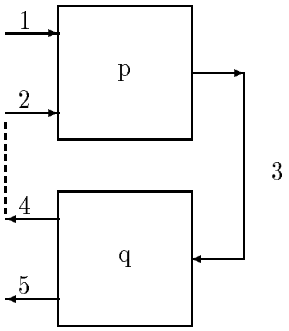
$$r; (\mathsf{Ap}_{A,\perp} \otimes \mathsf{id}_{A_6} \otimes \mathsf{id}_{A_7}) : A_1 \otimes A_2 \otimes A_4 \to I \otimes A_6 \otimes A_7.$$

By composing this with the canonical isomorphism $\lambda : I \otimes A \to A$ to remove the I, we obtain

$$s : A_1 \otimes A_2 \otimes A_4 \to A_6 \otimes A_7,$$

which is a morphism representing the desired network; in fact, $s = (p \otimes \mathsf{id}_A); q$. This procedure is the internalisation of composition in a $*$-autonomous category.

For an acyclic network such as the one above, we can always use composition to build the morphism we want. But if the network contains feedback loops, it is essential to use an Ap morphism to form the final link in each loop. For example, in the network



in which the dotted connection forms a cycle, we start with

$$p : A_1 \otimes A_2 \to A_3$$

and

$$q : A_3 \to A_4 \otimes A_5$$

and form

$$\Lambda(p; q); (\mathsf{Ap}_{A,\perp} \otimes \mathsf{id}_A); \lambda : A_1 \to A_5.$$

Note that although compact closure is necessary in order that all the types are formed using $\otimes$ rather than mixtures of $\otimes$ and $\parr$, it is not important that $(\cdot)^{\perp}$ is trivial.

Forming a feedback loop in a dataflow network is the way in which a recursive definition is made, and in the usual Kahn-style semantics [11] the behaviour of the resulting network is given by a least fixed point construction. By contrast, our constructions in **SProc** give a process representing the behaviour of

a network without explicitly introducing any least fixed points; the resolution of the recursive definitions is distributed among the interactions taking place at each point of connection between input and output. There is therefore some work to be done in order to see that our model gives the expected semantics. This question is dealt with in a forthcoming paper [6].

Another question is how to ensure that a network built from deterministic nodes remains deterministic, in light of the fact that in general, determinism is not preserved by composition in **SProc**. This is also dealt with in [6]; the property which *is* preserved is a combination of determinism and functionality, and we can check that all the nodes used in the modelling of SIGNAL satisfy this property.

4 Overview of the SIGNAL language

SIGNAL is one of a family of synchronous languages, along with ESTEREL, LUSTRE and SILAGE. In this context, *synchronous* means that outputs are produced simultaneously with the inputs which cause them. SIGNAL is a dataflow language and has been widely used in the areas of signal processing and control. A SIGNAL program is an executable specification of constraints expressed in terms of equations. Some of the equations specify that certain outputs are computed from certain inputs according to particular functions; this is familiar from any dataflow language. Other equations specify *relations* between signals. These relations are to do with constraints on the *clocks* of signals. The idea of a clock is shared with LUSTRE, but the way in which clock constraints appear is a characteristic feature of SIGNAL.

The concept of a clock arises when computations of different data streams are allowed to proceed at different rates. Rather than forcing a stream to consist solely of a sequence of genuine data values, undefined elements or delays are allowed. The clock of a stream indicates the times at which the stream contains a defined value. In LUSTRE, a stream consists of a sequence of genuine values and a clock which is represented by a stream of boolean values: t for a defined element of some stream and f for an undefined element. Thus the actual behaviour of a stream is obtained by "stretching out" its data values so that they appear at the times specified by its clock. In SIGNAL, by contrast, there is an "undefined" value $*$ which appears in streams at times when the clock is thought of as f. So the clock (in the LUSTRE sense) of a stream can be extracted from the stream itself.

The relations between clocks which can be expressed in a SIGNAL program are of the form "two streams have the same clock" or "stream A has a slower clock then stream B". Thus rather than completely defining the clock of every stream, as in a LUSTRE program, it is only necessary to express the essential timing constraints. The compiler then calculates the fastest clock on which everything can run.

5 SIGNAL programs as SProc processes

Our model of SIGNAL in **SProc** follows the general plan described previously. In order to represent clocks, a SIGNAL type A is modelled by the **SProc** type

$\delta\Delta A$. The $*$ action coming from the $\delta\Delta$ construction is used as the undefined value. When variables (names of streams) are declared, they are given initial values; this information is used in the construction of the processes modelling certain operators, as we will see later. Clock constraints are expressed by inserting extra processes into the network, which enforce the constraints.

6 SIGNAL operators in SProc

The simplest operators are the *static monochronous operators* which operate only on data and do not affect clocks. If $f : A \to B$ is a function then we have the stream extension $f^\omega : A^\omega \to B^\omega$; a static monochronous operator is a stream function of this form which can also ignore $*$ values. In **SProc** we have $f^\omega : A \to B$ defined by

$$f^\omega \quad = \quad \{((a, f(a)), f^\omega) \mid a \in \Sigma_A\}$$

and then the desired process which takes account of delays is just $\delta\Delta f^\omega$.

SIGNAL has a delay operator denoted by \$: the program fragment

$$Y = Z\$k$$

means that the stream Y is obtained by delaying the stream Z by k steps. Thus $y_n = z_{n-k}$ for $n \geq k$; for $n < k$ the values of y_n are taken from the sequence of initial values defined when Z is declared. To model this in **SProc** we use processes $\mathsf{sigdel}_a : \delta\Delta A \to \delta\Delta A$ for each type A and each $a \in \Sigma_{\delta\Delta A}$, where

$$\mathsf{sigdel}_a \quad = \quad \{((x, a), \mathsf{sigdel}_x) : x \in \Sigma_{\delta\Delta A}\}.$$

This deals with the case of a single delay; a longer delay can be built up from a series of single delays.

The remaining operators depend on clocks, and are known as *polychronous*. The when operator takes a signal X and a boolean signal B and outputs X when B is true. For example:

X :	1	4	$\perp$	5	$\perp$	6	...
B :	t	f	t	t	f	t	...
X when B :	1	$\perp$	$\perp$	5	$\perp$	6	...

To model this we introduce a boolean type $\mathbb{B}$ in **SProc** and use

$$\mathsf{when} : \delta\Delta A \otimes \delta\Delta\mathbb{B} \longrightarrow \delta\Delta A$$

where

$$\begin{aligned} \mathsf{when} \quad &= \quad \{((a, t, a), \mathsf{when}) : a \in \Sigma_{\delta\Delta A}\} \\ &\cup \quad \{((a, f, *), \mathsf{when}) : a \in \Sigma_{\delta\Delta A}\} \\ &\cup \quad \{((a, *, *), \mathsf{when}) : a \in \Sigma_{\delta\Delta A}\}. \end{aligned}$$

Next is the deterministic merge operator default. U default V takes its nth value from U if U is defined there, otherwise from V. This is modelled by

$$\mathsf{default} : \delta\Delta A \otimes \delta\Delta A \longrightarrow \delta\Delta A$$

where

$$\begin{array}{rcl} \mathsf{default} & = & \{((a,b,a),\mathsf{default}) : a,b \in \Sigma_{\delta\Delta A}\} \\ & \cup & \{((*,b,b),\mathsf{default}) : b \in \Sigma_{\delta\Delta A}\}. \end{array}$$

The event operator is used to extract the clock of a signal: it produces a boolean signal with value t when its input is defined, f otherwise. Its type is

$$\mathsf{event} : \delta\Delta A \longrightarrow \mathbb{B}$$

and

$$\begin{array}{rcl} \mathsf{event} & = & \{((a,t),\mathsf{event}) : a \in \Sigma_A\} \\ & \cup & \{((*,f),\mathsf{event})\}. \end{array}$$

To deal with clock constraints, first note that it is only necessary to take care of the case in which two signals are made to have the same clock; specifying that one clock is slower than another is a derived case. We use a process

$$\mathsf{synchro} : \delta\Delta A \otimes \delta\Delta B \longrightarrow \delta\Delta A \otimes \delta\Delta B$$

defined by

$$\begin{array}{rcl} \mathsf{synchro} & = & \{((*,*,*,*),\mathsf{synchro})\} \\ & \cup & \{((a,b,a,b),\mathsf{synchro}) : a \in \Sigma_A, b \in \Sigma_B\}. \end{array}$$

The way in which this works is that a process modelling a SIGNAL operator can cope with all possible clocks, so that outputs are non-deterministic to the extent that they contain many possible insertions of $*$s among the genuine actions. The process synchro passes on its inputs, but with the freedom of their clocks restricted so that both outputs have the same clock.

We now present a small example to illustrate composition and some of the SIGNAL operators. We use Abramsky's "stream of consciousness" tables to show possible traces of processes. The first table is for when.

X	B	X when B
1	t	1
4	f	$*$
$*$	t	$*$
5	t	5
$*$	f	$*$
6	t	6
$\vdots$	$\vdots$	$\vdots$

We aim to form the composition when; event. Two possible traces for event are shown below.

Y	event$Y$
$*$	f
1	t
2	t
4	t
$*$	f
5	t
$\vdots$	$\vdots$

Z	event$Z$
1	t
$*$	f
$*$	f
5	t
$*$	f
6	t
$\vdots$	$\vdots$

To find a possible trace for the composition, we need traces in which the output of when matches the input of event. The second trace of event shown above fits with the trace of when shown, and together these give a trace of

$$\mathsf{when}; \mathsf{event} : \delta\Delta A \otimes \delta\Delta\mathbb{B} \longrightarrow \delta\Delta\mathbb{B}.$$

The result is a boolean signal which is *not* quite the clock of the signal X.

X	B	X when B	Z	event$Z$ ($Z = X$ when B)
1	t	1	1	t
4	f	$*$	$*$	f
$*$	t	$*$	$*$	f
5	t	5	5	t
$*$	f	$*$	$*$	f
6	t	6	6	t
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

7 Conclusions and Future Work

We have shown how to interpret SIGNAL operators and programs in **SProc** in a uniform, typed fashion. The task is made straightforward by the synchronous and relational nature of SIGNAL. Even though it is a simple application, this is a worthwhile task as SIGNAL is already used in large-scale applications in the field of digital signal processing. We have been able to model a SIGNAL program for a digital stopwatch in **SProc**, the details of which do not discuss here due to space restrictions. The use of delay monads to model clocks suggest that we have the right kind of structures in our framework for modelling such languages. This feature is brought out more clearly in modelling LUSTRE [7] where there is a strong interplay between the types and the delays; in SIGNAL

this is reflected in the clock calculation. Each SIGNAL fragment can operate on its own clock and the SIGNAL compiler uses the dependencies to construct a hierarchy of clocks. If a single master clock exists (the *coarsest common clock*) it will be derived by the compiler. We aim to show that such a clock derived via our model of a SIGNAL program in the Interaction Category framework is the same as that derived by the compiler.

We also plan to investigate the translation of ESTEREL which is similar in spirit to SIGNAL but has more powerful constructs. There is also work in progress in which we are trying to capture processes directly in **SProc**.

Acknowledgements

We would like to thank Samson Abramsky for being the guiding light. The referees Lindsay Errington and Tom Maibaum provided useful comments. Simon Gay is funded by an SERC Studentship; and Raja Nagarajan by project CONFER (ESPRIT BRA 6454) and ICCF. We gratefully acknowledge their support.

References

[1] S. Abramsky. Interaction categories. In this volume.

[2] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. Technical Report DoC 92/24, Department of Computing, Imperial College of Science Technology and Medicine, 1992.

[3] P. Aczel. *Non-well-founded sets*. CSLI Lecture Notes 14. Center for the Study of Language and Information, 1988.

[4] G. Berry, P. Couronné, and G. Gonthier. Synchronous programming of reactive systems: An introduction to ESTEREL. Technical Report 647, INRIA, 1986.

[5] G. Berry and P.-L. Curien. Theory and practice of sequential algorithms: the kernel of the applicative language CDS. In J. C. Reynolds and M. Nivat, editors, *Algebraic Semantics*, pages 35–84. Cambridge University Press, 1985.

[6] S. J. Gay. On iteration and interaction. Draft paper, 1993.

[7] S. J. Gay and R. Nagarajan. Modelling LUSTRE in Interaction Categories. Paper in preparation, 1993.

[8] P. le Guernic, T. Gautier, M. le Borgne, and C. le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[10] P. N. Hilfinger. Silage, a high-level language and silicon compiler for digital signal processing. In *IEEE Custom Integrated Circuits Conference CICC–85*, pages 213–216. IEEE, May 1985 1993.

[11] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, 1974.

[12] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.