The SynDEx software environment for real-time distributed systems design and implementation

C. Lavarenne, O. Seghrouchni, Y. Sorel, M. Sorine
INRIA-Rocquencourt - Domaine de Voluceau
B.P.105 – 78153 LE CHESNAY Cedex

Proceedings of the European Control Conference (July 1991).

Abstract. Synchronous languages propose a new way to specify the reactions of a real time system to its environment: they are described within the language and no more at the operating system level. Furthermore the reactions of the Run Time Support (RTS) to events have also to be specified. RTS give access to processor resources. Predictability and efficiency are now requested. SynDEx has been designed to generate RTS for programs written in the synchronous language SIGNAL, target computers being distributed processors. The target performances are taken into account to automatically partition the program over the processor set and to schedule executions in order to maximize the global performance. We present SynDEx and the specification formalism.

Keywords. Realtime programming, SIGNAL, Run Time Support, program partitionning and scheduling, distributed multiprocessor, formal specification.

1 INTRODUCTION

Emerging synchronous languages propose a new way to specify the reactions of a real time system to its environment: concurrency, communication and events handling are described within the language and no more at the operating system level. Furthermore the reactions of the Run Time Support (RTS) to events have to be specified at the programming level or hidden. This allows to consider that the programs have instantaneous reaction to external events (no extra events are generated by the RTS), making easier verification and improving program readability and portability. Nevertheless, RTS are still usefull to give access to processor resources (time, memory, i/o channels...). Predictability and efficiency are now requested. The SynDEx software environment has been designed to generate RTS for programs written in the synchronous language SIGNAL, target computers being distributed processors. An RTS resulting from a given SIGNAL program and a target multiprocessor has a formal specification allowing verification and code generation. The target performances (speed of processors, throughput of i/o channels, memory size) are taken into account to automatically partition the program over the processor set and to schedule executions in order to maximize the global performance. We present and demonstrate the current prototype of SynDEx and the specification formalism.

2 SynDEx AND SIGNAL

2.1 The PST$\otimes$ formalism

We follow, when possible, the notations of [1] §3.

Values Let A be a finite set of names, called ports. $\forall a \in A$, $\mathcal{D}_a$ is the domain of values carried by a. Four values are distinguished: tt (true), ff (false), $\top$ (present), $\perp$ (absent). $\mathcal{B}_{tt} = \{tt, ff\}$ and $\mathcal{B}_\top = \{\top, \perp\}$ are, in an obvious way Boolean algebras. $\mathcal{D}_A = \cup_a \mathcal{D}_a$ will be sufficient in the sequel and called $\mathcal{D}$ when A is given by the context. We still need: $\mathcal{D}_{A\top}$ (or $\mathcal{D}_\top$) $= \mathcal{D}_A \cup \{\top\}$, $\bar{\mathcal{D}}_A$ (or $\bar{\mathcal{D}}$) $= \mathcal{D}_A \cup \mathcal{B}_\top$.

Time and traces We now use some basic facts about Pomsets (see [2], [3]). To each $a \in A$ is associated a labeled partial order (ℓpo), that is a 4-tuple $(\pi_a, \bar{\mathcal{D}}_a, \leq_a, \ell_a)$, where: π_a is a vertex set, modeling instants, $\bar{\mathcal{D}}_a$ is defined as before, $\leq_a$ is a partial order on π_a ($t_1 \leq_a t_2$ means: event (or instant) t_1 is preceding t_2), ℓ_a is a labeling function assigning values to instant-vertices.

Identities of instants are not important, so that ℓpo's are only to be known up to a ℓpo isomorphism. We have the following equivalence: $(\pi_a, \bar{\mathcal{D}}_a, \leq_a, \ell_a) \cong (\pi'_a, \bar{\mathcal{D}}_a, \leq'_a, \ell'_a)$ iff $\exists f$ bijection $\pi_a \to \pi'_a$ such that $t_1 \leq_a t_2 \Leftrightarrow f(t_1) \leq'_a f(t_2)$ and $\ell_a(t) = \ell'_a(f(t))$, $\forall t \in \pi_a$. The equivalence classes, denoted $\theta_a = [\pi_a, \bar{\mathcal{D}}_a, \leq_a, \ell_a]$ are Pomsets and look like generalized traces. More precisely we say that θ_a is sequential if $\ell_a^{-1}(\mathcal{D}_{a\top})$ is totally ordered (property independent of the chosen representative of θ_a). θ_a will be called a trace of a and an S-trace if it is sequential.

Remark that traces allow autoparallelism: when needed, sequentiality will have to be explicitely imposed (e.g. for input/output parts of a delay operator). Sometimes it is not needed (e.g. for the SIGNAL process $a := b + c$) at a level of description, but needed at an other level (e.g. for the "hardware adder" $a := b + c$).

Clocks A clock (resp. an S-clock) is a trace (rep. an S-trace) with values in $\mathcal{B}_\top$. We have the following basic partial order on the set $\mathbf{K}$ of clocks: $H_1 \subseteq H_2$ iff $h_1^{-1}(\top) \subseteq h_2^{-1}(\top)$ for some $h_1 \in H_1$, $h_2 \in H_2$ (that $\subseteq$ is a partial order is easy to prove). $S\mathbf{K}$ is the set of S-clocks.

Operations on traces

Parallel composition of traces (or concurrence): Let $\theta_i = [\pi_i, \bar{\mathcal{D}}_i, \leq_i, \ell_i]$ for $i = 1, 2$, then $\theta_1 \| \theta_2 = [\pi_c, \bar{\mathcal{D}}_c, \leq_c, \ell_c]$ with $\pi_c = \{1\} \times \pi_1 \cup \{2\} \times \pi_2$, ($\times$ is the cartesian product of sets), $\bar{\mathcal{D}}_c = \bar{\mathcal{D}}_1 \cup \bar{\mathcal{D}}_2$, $(i, x) \leq_c (j, y)$ iff $i = j$ and $x \leq_i y$, $\ell_c((i, x)) = \ell_i(x)$.

Let $\varepsilon = [\emptyset, \emptyset, \emptyset, \emptyset]$ be the empty trace. It is easy to show [3]: $\theta_a \| \varepsilon = \varepsilon \| \theta_a = \theta_a$; $\theta_a \| \theta_b = \theta_b \| \theta_a$; $\theta_a \| (\theta_b \| \theta_c) = (\theta_a \| \theta_b) \| \theta_c$.

Sequential composition of traces: $\theta_1 ; \theta_2 = [\pi_S, \bar{\mathcal{D}}_S, \leq_S, \ell_S]$ with $\pi_S, \bar{\mathcal{D}}_S, \ell_S$ as $\pi_c, \bar{\mathcal{D}}_c, \ell_c$ above and $(i, x) \leq_S (j, y)$ iff $i = j$ and $x \leq_i y$ or $i = 1$ and $j = 2$.

We have: $\theta_a ; \varepsilon = \varepsilon ; \theta_a = \theta_a$ and $(\theta_a ; \theta_b) ; \theta_c = \theta_a ; (\theta_b ; \theta_c)$.

Orthogonal composition of traces (or orthocurrence): $\theta_1 \times \theta_2 = [\pi_1 \times \pi_2, \bar{\mathcal{D}}_1 \times \bar{\mathcal{D}}_2, \leq_1 \times \leq_2, \ell_1 \times \ell_2]$. ε is also a left and right neutral element, $\times$ is associative.

Processes: they are sets of traces. Examples (Kleene star and Dagger): $\theta_a^* = \{\varepsilon,\ \theta_a,\ \theta_a ; \theta_a,\ \theta_a ; \theta_a ; \theta_a, \ldots\}$, $\theta_a^\dagger = \{\varepsilon,\ \theta_a,\ \theta_a \| \theta_a,\ \theta_a \| \theta_a \| \theta_a, \ldots\}$. Previous operations can be extended to processes: the resulting process is the set of resulting traces associated to each pair of traces. Usual set operations are also valid for processes.

Traces and processes homomorphisms. Preceding operations act independently on events ; domain values, order relation and labeling of traces. We need to express dependences between resulting events and initial events and values.

Let θ be a trace on $\bar{\mathcal{D}}_\theta$ (i.e. $\theta = [\pi_\theta, \bar{\mathcal{D}}_\theta, \leq_\theta, \ell_\theta]$) and f be a function mapping $\bar{\mathcal{D}}_\theta$ to traces on $\bar{\mathcal{D}}_\theta$. f can be extended to a function mapping traces on $\bar{\mathcal{D}}_\theta$ to traces on $\bar{\mathcal{D}}'_\theta$ in the following way:

$$f(\theta) = \|_{\substack{s,t \in \pi_\theta \\ s \leq_\theta t}} f(\ell_\theta(s)) ; f(\ell_\theta(t))$$

(In order to avoid extra parentheses, we give to ; a higher precedence than to other binary operations). Events, domain of values and labeling function of $f(\theta)$ are those of $\|_{t \in \pi_\theta} f(\ell_\theta(t))$, the order relation is the one of this last trace augmented with $\cup_{s<t} \left(\pi_{f(\ell_\theta(s))} \times \pi_{f(\ell_\theta(t))}\right)$: f can still be extended in the obvious way to processes.

Application 1: memoryless functional processes We give the semantics of a process F with input ports $I_0, \ldots, I_{m-1}$, output port O, computing "repeatedly" $f(x_0, \ldots, x_{n-1})$, $x_i \in \bar{\mathcal{D}}$. Denoting in the same manner a quantity a and the trace $[\{a\}, \{a\}, \{(a, a)\}, \{(a, a)\}]$ we have:

$$\begin{cases} F = \tilde{f}((\bar{\mathcal{D}}^n)^\dagger \times (I_0 \| I_1 \| \cdots \| I_{n-1}) ; O) \\ \text{with} \\ \quad \tilde{f}(x_0, \ldots, x_{n-1}, I_j) = (x_j, I_j) \\ \quad \tilde{f}(x_0, \ldots, x_{n-1}, O) = (f(x_0, \ldots, x_{n-1}), O) \end{cases}$$

(where for example I_0 is a port name and a trace, (x_j, I_j) a label and a trace ...). Argument traces are here finite occurences of n-tuples on $\bar{\mathcal{D}}$. F is called a $\bar{M}$-function and denoted $f(x_0, \ldots, x_{n-1})$. We can augment order by replacing the Dagger by the Kleene star, so that results are also finite sequences. This is too restrictive for applications in view (ordering events labeled by $\perp$ is useless in some cases) so we prefer to define sequential memoryless function (or functional processes), in short $S\bar{M}$-functions, as functions mapping S-traces to S-traces. A sufficient condition for F to be a $S\bar{M}$-function is F being a $\bar{M}$-function, extension of some f satisfying:

$$\exists i_0,\ x_{i_0} = \perp \Rightarrow \forall x_i \in \bar{\mathcal{D}},\ i \neq i_0,\ f(x_0, \ldots, x_{n-1}) = \perp$$

This condition is also necessary for $n \leq 1$.

Example : the memoryless part of the PST⊗ Kernel. Let $\mathcal{D}$ and $\bar{\mathcal{D}}$ as above and define the following functions on $\bar{\mathcal{D}}$ with singleton value traces:
$P : x \to \top$ if $x \in \mathcal{D}$, else $\perp$ (value presence function)
$S : x \to \top$ if $x \in \mathcal{D}_\top$, else $\perp$ (synchro event presence functions)
$T : x \to \top$ if $x = tt$, else $\perp$ (truth function).

Substituting in the definition of $\bar{M}$-functions, P (resp. S, T) for f with $n = 1$, leads to $S\bar{M}$-functions still called P (resp. S, T).

In the same manner we can extend booleans operations of $\mathcal{B}_\top$ to operations on clocks, so that $\mathbf{K}$ is a Boolean Algebra with operations denoted $., +, \neg$. We remark that . is a $S\bar{M}$-function, but $+$ and $\neg$ are not so (union or complement of totally ordered sets are not necessarily totally ordered). Let $\bar{\mathcal{D}}_1$ and $\bar{\mathcal{D}}_2$ be two domains of values and $\otimes$ be the function $(x_1, x_2) \in \bar{\mathcal{D}}_1 \times \bar{\mathcal{D}}_2 \to (x_1$ if $x_1 \in \mathcal{D}_1$ and $x_2 \in \mathcal{D}_{2\top}$ else $x_2) \in \bar{\mathcal{D}}_1 \cup \bar{\mathcal{D}}_2$. $\otimes$ can be extended into an $S\bar{M}$-function of traces called the modulation product.

Application 2 : memoryless functional relation It will be sufficient to consider relations of the form $\{\theta_0, \ldots, \theta_{n-1}) \mid f(\theta_0, \ldots, \theta_{n-1}) = \theta\}$ where f is an $S\bar{M}$-function and θ a given trace. Such a relation is in fact a process. We call it a $\bar{M}$-relation.

The PST⊗ Kernel It possesses the following operations and relations defined on traces:
i) $S\bar{M}$-functions with in particular $P, S, T, \otimes$ and clock multiplication
ii) $\bar{M}$-relations and clock addition and negation
iii) Constructors $\|, ;, \times$.
iv) $P_A !! A'$ with $A' \subseteq A$ sets of port names.

The last operation is the restriction of P_A, a set of traces associated with ports of A, to traces associated with ports of A' (SIGNAL notation).

It will be convenient to denote by **NK** the set of discrete sequential clocks, so that θ is discrete sequential iff $S(\theta) \in \mathbf{NK}$.

2.2 The PST$\otimes$ based model of SIGNAL

Signals Consider the following equivalence relation on traces: $\theta_1 \equiv \theta_2$ iff $\theta_1 \otimes P(\theta_1) = \theta_2 \otimes P(\theta_2)$ (θ_1 and θ_2 differ only by $\top$ or $\bot$ labeled events).

Signals are $\equiv$ equivalence classes of discrete S-traces. We write $s = [\theta]$. Each clock is equivalent to $\bot$. The clock of a signal $s = [\theta]$ is $P(\theta)$, we write instead $P(s)$. A process is a set of signals.

A model of SIGNAL Such model is a list constraints on discrete S-traces (and then on signals through $\equiv$):
i) $R(x_1, \ldots, x_p) = r(x_1, \ldots, x_p) \| $ synchro $(x_1, \ldots x_p)$ where r is a $\bar{M}$-relation and synchro $(x_1, \ldots, x_p) = (P(x_1) = \cdots = P(x_p))$
ii) x \$ 1 init $x_0 = x_0$; x.
iii) x when $b = x \otimes (P(x).T(b))$
iv) u default $v = u \otimes (v \otimes (P(u) + P(v)))$
v) $| = \|$
vi) $P!!x_1, \ldots, x_p$ restriction to the listed set of signals.

This model can be proved to be equivalent to the (discrete sequential) trace model of [1].

2.3 SynDEx / SIGNAL interface

SIGNAL and PST$\otimes$ programs Using the preceding model, SIGNAL programs can be rewritten into PST$\otimes$ programs acting upon discrete S-traces satisfying $\theta = \theta \otimes P(\theta)$, or equivalently upon traces satisfying the relation $P(\theta) = S(\theta) \in \mathbf{NK}$. Other clock relations come out of PST$\otimes$ translations of type i) instruction (the synchro part), type ii) $P(y) = P(x)$, type iii) and iv), by using the identities $P(\theta_1 \otimes \theta_2) = (P(\theta_1) + P(\theta_2)).S(\theta_2)$, $S(\theta_1 \otimes \theta_2) = S(\theta_2)$.

The resulting clock relation (after $\|$ composition) is normally solvable by the SIGNAL compiler if the program is "well written". The solutions are $S\bar{M}$-functions of external clocks (visible clocks after !! operation and free clocks resulting from non unicity of solutions) and possibly still $\bar{M}$-relations involving external clocks and input boolean signals (i.e. signals on $\mathcal{B}_{tt}$ associated with an external input port)(see [4]). This set of solution clocks may be partially ordered using $\subseteq$ and the following property: $H_1 \subseteq H_2$ iff $\exists H, H_1 = H_2.H$ or $\exists H', H_2 = H_1 + H'$.

Beside clock relations, the SIGNAL compiler produces a Dependences Graph conditioned by clocks (the Conditional Dependences Graph, CDG for short), union of CDG associated to SIGNAL "instructions" except ii) [4]. The PST$\otimes$ program can be treated in a different way, we sketch now.

Structuring PST$\otimes$ programs We present the objectives and principles of this structuring. In order to obtain runable programs, at least in well defined contexts, we need to structure the global trace relation $\mathcal{P}$ represented by the program into $\mathcal{P} = hboxRext \| F$ where Rext is a $\bar{M}$-relation on external boolean traces or clocks defining an admissible context and F is an homomorphism mapping external input traces to internal and external output traces. Rext has to be as "simple" as possible. Simplifications are obtained by boolean manipulations and clock $\subseteq$-ordering. For example the property $H \subseteq H'$ and $H' \in \mathbf{NK} \Rightarrow H \in \mathbf{NK}$ leads to write sequentiality constraints only for some maximal clocks (not all, if this constraint has been kept only for traces operated by a delay, in order to be less rstrictive). Structuring the "functional part" F of the program leads to:

1) Substitute the resulting expressions of clocks (after solving and simplifying) for their instances in F and again to simplify. For example $y = x \otimes (P(x).T(b))$ and $T(b) \subseteq P(x)$ leads to $y = x \otimes T(b)$.

2) Organize the program "around memories": for each delay operator a **NK** clock has been determined (common clock to input and output traces). All the delay operators with the same clock Φ_i, $i = 1, \ldots, m$, can be rewritten as a single delay, mapping vector valued traces x'_i to x_i: $x_i := x_{i_0}; x'_i$. x'_i with $S(x'_i) = \Phi_i$ is necessarily the output of a $\bar{M}$-function which can be written as $x'_i := f_i(x_i, u_i, a_i) \otimes S(a_i)$ where a_i is the vector valued trace whose components are θ_p for all the ports p such that: $S(\theta_p) = \Phi_i$ and there is a direct dependence between p and the input port of the delay associated to Φ_i. u_{i1} is defined in the same manner without the clock condition, so that all direct dependences are taken into account. Let y_i be the vector valued trace whose components are θ_p for all the ports not considered until now, such that $S(\theta_p) = \Phi_i$. y_i may depend upon x_i, a_i and u_{i2}, to be complete. Set $u_i = u_{i1} \| u_{i2}$. We can write $y_i := h_i(x_i, u_i, a_i) \otimes S(a_i)$ for some $\bar{M}$-function. The introduced vector valued traces are associated to ports of the form $p_1 \| \cdots \| p_m$ where p_i was associated to the ith component. This leads to a candidate intermediate code to link the SIGNAL and SynDEx environments.

The proposed SynDEx / SIGNAL interface A general input program for SynDEx may have the form $\mathcal{P} = hboxRext \| F_1 \| \cdots \| F_k$ where F_i is an input/output mapping admitting the "state representation":

$$F_i(u_i, a_i) = (x'_i := f_i(x_i, u_i, a_i) \otimes S(a_i)$$

$$\| x_i := x_{i_0}; x'_i$$

$$\| y_i := h_i(x_i, u_i, a_i) \otimes S(a_i)) !! y_i$$

where k is the number of clocks. It may happen that x_i be constant, corresponding to cases where $S(a_i)$ is not associated to a delay operator: F_i reduces to a $\bar{M}$-function. u_i, a_i may be, for some components external input traces or other F_j outputs. Remarking that $S(y_i) \subseteq S(a_i)$ the clock hierarchy is still apparent on the automata network $F = \|_i F_i$.

In fact, $\mathcal{P}$ appears as an expansion of a trace $\tilde{\mathcal{P}}$ where events are no more instants but clocks: $\tilde{\mathcal{P}} = [\mathbf{K}_F, \mathcal{D}_{I0}, \leq_{\mathbf{K}_F}, \{(\Phi_i, F_i) \mid \Phi_i \in \mathbf{K}_F\}]$ where $\mathbf{K}_F$ is the finite set of clock, $\leq_{\mathbf{K}_F}$ the clock ordering and $\mathcal{D}_{I0}$ the union of input-output domains of the F_i''s satisfying Rext.

3 MULTIPROCESSOR PROGRAMMING IN SynDEx

The labels F_i in $\tilde{\mathcal{P}}$ are input-output functions associated to some events, the clocks. This vision is not precise enough if, for example, we want to make the data flow aspect of the program apparent: we have to change the level of atomicity and we are again to $\mathcal{P}$-level. Each event is viewed as a multievent and F_i as actions $\{f_i, h_i, ;\}$ to perform at each elementary event (the synchronous instants). Again this level of description may necessite a refinement if we want to detail the different steps when performing computation or communication. We modelize again this refinement, as a trace homomorphism: we map labels of $\mathcal{P}$ (i.e. $f_i, h_i, ;$) to traces representing execution schemes. The event sets of these traces are finite sets, the labels contain informations on execution time and memory requirement. Allocation of a process P_i to a processor p_k can be represented by $\delta_{ij} = P_i \times p_k$ if $j = k$ else ε. A scheduling is a local (to a processor) linear augment of the allocated program $\|_{i,j} \delta_{i,j}$, a linear argument θ' of a trace θ differing from θ only in its partial order: $\leq_{\theta'}$, is a linear superset of $\leq_\theta$. Scheduling have to be compatible (they have to constitute an allowed global scheduling). This leads to the problem of determining a new level of atomicity where atoms can be a priori linear augmented [4]. The structure of $\mathbf{K}_F$ is important for the allocation/scheduling problem. The actual optimizer considers only the case $|\mathbf{K}_F| = 1$, allowing off line optimization. For $|\mathbf{K}_F| > 1$, on line scheduling is requested and will be studied later.

4 THE COMMUNICATION KERNEL

Usually a real time distributed executive provides services such as time and resources management (memory, files), allocation and scheduling of processes and inter-processes communication and synchronization.

Among these services, one of the most important is the communication support. We emphasize this issue because of the peculiar role of inter-processor communications in distributed memory multi-processors. Access to communication resources leads to the well known bottleneck. In order to overcome this problem, we intend to map as well as possible the application graph algorithm on the multi-processor target graph: inter-processor communications and execution time are minimized. The study of these graphs allows the allocation and scheduling of processes and communications in a static way before execution.

Automatic generation of executives does not suppose a system background process resident on every processor and invoked by every process requiring a communication service. SynDEx generates the minimum amount of system processes supporting inter-processor communications, the executive will be built using a reduced executive kernel.

4.1 Communication services

Let's review briefly the main issues involved in a end-to-end communication (message passing scheme), referring to OSI communication services.

Datalink layer : An application process performing a communication in order to deliver a data to an other application process placed on an other processor, must send messages through the network. The messages travel on routes (network path), they are exchanged from processors to nearest processor through a physical connection (bus, link) and finally reaches the receiver processor.

Network layer : Any process in a processor can send a message to any other process located in any processor, providing end-to-end transmission. Forwarding through intermediate processor on a route is transparent to the application processes.

Transport layer : End-to-end transmission of messages through the network must be reliable, they are guaranted not to congest, block or corrupt the communication network. Communication protocols prevent from these misfunctionings. An end-to-end communication can be processed using a more or less strong synchronization protocol. A synchronous protocol, where sender and receiver application processes must cooperate, is a strong synchronization. For example in a data-driven mode, the sender transmits data only when the receiver has acknowledged the previous data. Furthermore a higher protocol can be added which carries out routes arbitration avoiding dead-locks.

4.2 Executive kernel

In order to achieve the previous functionalities, we have defined an executive kernel described below.

PE and PR : At one end of the communication path, the PE object (sender gate) builds a *data message* with the raw data produced by application process and routing informations specifying the PR lo-

cation to reach. When route arbitration is required, before sending data, the PE has to ask for a route. Then it builds a *control message* with the control operation type (request or release) and routing information specifying the route arbiter to reach. The messages are sent to the RT object described below. At the other end, the PR object (receiver gate) extracts from the message raw data to be consumed by the application process. If a synchronous protocol is used, it sends an acknowledge control message to the sender.

PES : This object (input/output gate) performs input and output of messages from and to other processors through the communication network. Actually it is used when messages are forwarded. In both directions, messages are buffered in a FIFO mode. This allows to support several sequentialized communications. PES have access the network. For example when a message goes through a bus in order to reach the neighbouring processor, a PES sends a request control message to the bus arbiter, waits for a grant message, then sends its data message and finally sends a release message to the bus.

RT : A router object is located in every processor. It receives messages directly from the PE's and, using a looktable, it determines which processor has to be reached. The message is sent to the proper PES (used as output) in charge of its delivery on the network. It receives from PES's (used as input) messages sent by PE's and PR's located in distant processors. Using the routing information inside the message and a looktable, it determines if the message is reaching the proper processor, in that case it sends it to the PE, in the other case it forwards the message to the proper PES (used as output).

ARB : An arbiter object is located in every processor. It receives requests from PE's and from PES's and decides which of them must be satisfied and finally sends grants. In order to achieve this, it refers to scheduling informations. In both cases the arbitration policy can be choosen by the user or given by SynDEx. More information can be found in [6].

5 PROTOTYPE DESCRIPTION

The SynDEx programmer describes the multiprocessor as a *hardware graph* and the application as a *software graph*, then maps the two graphs by specifying *placement constraints* and by running a *placement algorithm* and finally runs a *code generator* which produces the *multi-processor executive* to be compiled and linked with system and user libraries.

A SynDEx browser picture is given in figure 1 as it appears on a workstation screen showing in this example an adaptative equalizer mapped on a bi-processor. The user needs to interact with only one window divided in five areas, each with its own pop-up walking menu(s). SynDEx commands may be invoked either

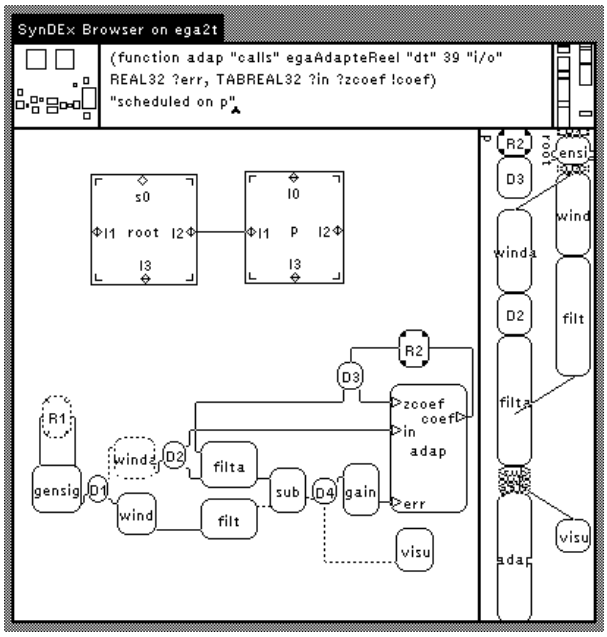

Figure 1: SynDEx browser

from menus or from a text interpreter.

5.1 Topography area and its zoom

The bottom-left area allows the user to edit the *topography* of the hardware and software graphs. The *topography area* may bee magnified under the control of the top-left area. This *zoom area* displays a schematic map of the topography area, where only boxes frames are drawn.

In the following, a graph vertice will be called a *node*, and a graph edge a *connection*. For the hardware graph, nodes represent processors (memory and computation capacity) or buses (shared communication capacity) and connections represent bidirectional data links between hardware nodes. For the software graph, nodes represent computation activities and connections represent oriented data transfers.

Each displayed object has its own menu of commands that the user may pop-up by depressing a mouse button when one of the object's *remarquable point* (box corners and center and broken line vertices) is highlighted (at any moment, that remarquable point which is nearest to the mouse pointer is highlighted).

Any remarquable point may be moved by dragging it with the mouse. The user may also collect several nodes (by shift-clicking each one) into a *multiple selection* which is a kind of *hyper-node* with an invisible box which may be used to move, resize, copy or cut all together the members of the multiple selection.

5.2 Text area

The top middle area is used for text input and output. The text area of figure 1 shows the SynDEx command which was input to build the software node **adap** (see

[5] for full syntax reference). To input a command, the user may type it entirely or just modify a template obtained from the text area menu (the template comes with comments for help). To execute a command, the user selects its text and runs the SynDEx interpreter from the text area menu. The SynDEx interpreter inserts explicit error messages as highlighted text before the character where the error was found, giving the user an easy way to locate and correct the error.

5.3 Schedule area and its zoom

When the user has completed the hardware and software graphs in the topography area, he runs a placement algorithm from the text area menu and the resulting *schedule* is displayed in the bottom-right area. This area may be magnified under the control of the top-right area in the same way the topography area is controlled by the top-left area.

There is one column per processor, with time running from top to bottom. Each activity is displayed as a box which height is proportionnal to the activity's duration. Each inter-processor communication is displayed as a diagonal line from the sender processor column to the receiver processor column with vertical projection proportional to the communication's duration. Inter-processor communications appear stippled in the topograhy area.

5.4 Placement algorithm

The placement algorithm presently implemented in SynDEx is a greedy algorithm based on an heuristic looking for critical path minimization [7]. It assumes that the hardware graph is *homogeneous* (all processors are of the same type), that the software graph is an *unconditioned data-flow graph* and that graph cycles may only be done through *delays*. The user may constrain activities to be placed on a given processor by collecting them into a multiple selection and by running the **placement** command from the processor's menu. In figure 1, **R1** and **winda** appear stippled to show they are constrained.

Step 1 : an initial set of candidates for placement is made of activities with no predecessors (and of *delays* in order to cut cycles).

Step 2 : among all the possible pairs *candidate-processor*, the one which minimizes a given *cost function* is selected and its candidate is placed on its processor.

Step 3 : the candidate just placed is removed from the candidates set and among its successors (except *delays*), those having all their predecessors already placed are added to the candidates set.

Step 4 : steps 2 and 3 are repeated until the candidates set is empty.

The *cost function* in step 2 takes into account the candidate's *schedule flexibility* [8] and its volume of inter-processor communications, would it be placed on the processor. The *schedule flexibility* is the difference between the candidate's *latest* and *earliest* dates. An activity's *earliest start* date is the maximum of its predecessors' *earliest end* dates. An activity's *latest end* date is the minimum of its successors' *latest start* dates. The difference between an activity's *end* and *start* dates is the activity's *duration*, given by the user when building each activity. Initial earliest and latest dates are computed in step 1 so that activities located on the software graph critical path(s) have a null schedule flexibility. As activities are placed, sequentialization of parallel activities and/or inter-processor communications diminish schedule flexibilities by delaying earliest dates. Furthermore, as inter-processor communications share hardware links, the more their volume, the more they will delay following inter-processor communications.

Therefore the best processor for a candidate will be either the one it was constrained to be placed on or the one giving it the most schedule flexibility and the minimum communication volume (i.e. minimize the difference between the last and the first), and the candidate to place first will be the one which has, with its best processor, the least schedule flexibility and the least communication volume (i.e. minimize the sum of the two).

When an activity is placed, each inter-processor communication duration is kept to a minimum by choosing the fastest *route* (hardware graph path) between the sender and receiver processors. If the hardware graph happens to be not connex, a hardware connection is automatically built as close as possible to the sender and receiver processors to ensure that at least one route connects them. This is usefull when the user lets the algorithm build its own idea of the machine best suited for the application. To do so, the user reduces the hardware graph to a single processor before running the placement algorithm. Then, before step 1, the algorithm prompts the user for a number of processors with an initial answer equal to the upper bound of the ratio between the activities durations sum and the critical path length. Then the algorithm connects that number of processors as it runs.

5.5 Code generation

All code generators (SynDEx, postscript, executive) and other file interface commands are run from the top-left area menu.

The *SynDEx code generator* produces a backup file containing SynDEx commands describing the hardware and software graphs and their mapping after a placement. This file may be later loaded to restore the SynDEx browser in the saved state.

The *postscript code generator* produces more detailed output than that available on the display screen. This output may be used for hardcopy or insertion into another document.

The *executive code generator* produces the source code necessary to run the described application on the described multi-processor as it has been scheduled. The code calls external user (activities) and system (executive kernel) libraries, allocates needed buffers, network addresses and routing tables ..., all with cross-references and comments refering to the original SynDEx commands specifications. To ease the final compilation process, a *makefile* is also generated. SynDEx has been designed to generate executives for different langages though it presently generates only OCCAM code for the Inmos Transputer Development Toolset.

6 CONCLUSION

Partial order programming has been used as a common formal framework to modelize SIGNAL programs and the various SynDEx transforms leading to multiprocessor implementation. This formalism should be adequate for further studies. The actual prototype of SynDEx has been presented. It has been already successfully used in large scale signal processing applications.

References

[1] A. Benveniste, P. Le Guernic, Y. Sorel, M. Sorine : *A denotational theory of synchronous reactive systems.* Information and Computation (1990).

[2] V. Pratt : *Modeling concurrency with partial orders.* International Journal of Parallel Programming. Vol. 15 n^o 1 (1986).

[3] U. Engberg : *Partial orders and fully abstract models for concurrency.* Ph D Thesis Aarhus University (1990).

[4] C. Figueira, T. Gautier, B. Le Goff, P. Leguernic : *Towards multiprocessor implementation of real time data flow program* in W.W. Wadge, Editor, ISLIP'88 (1988).

[5] C. Lavarenne, Y. Sorel : *SynDEx, un environnement de programmation pour multi-processeur de traitement du signal – Manuel de l'utilisateur version v.0.* Rapports Techniques INRIA n°113 (1989).

[6] N. Ghezal, S. Matiatos, P. Piovesan, Y. Sorel, M. Sorine : *SynDEx, un environnement de programmation pour multi-processeur de traitement du signal – Mécanismes de communication.* Rapports de Recherche INRIA n°1236 (1990).

[7] N. Ghezal : *Quelques méthodes de répartition et d'ordonnancement de processus de traitement du signal sur un réseau de transputers.* Doctorat thesis, Paris-Sud Orsay University (1989).

[8] Z. Liu, J. Labetoulle : *A heuristic method for loading and scheduling flexible manufaturing systems.* Proc. of the Int. Conf. Control 88, London, IEE Conference Publication no 285 p 195–200, (1988).