Analyse et Conception à Objets Temps Réel
pour Implantation asynchrone/Synchrone
(ACOTRIS)

SIGNAL → SynDEx translation V1.0

Loïc Besnard[1] Thierry Gautier[2] Qiuling Pan[3]
Christophe Macabiau[4] Yves Sorel[5]
July 4, 2002

CS SI
MBDAM
CEA/LIST
INRIA
SITIA

[1]IRISA/CNRS, e-mail: Loic.Besnard@irisa.fr
[2]IRISA/INRIA, e-mail: Thierry.Gautier@irisa.fr
[3]IRISA/INRIA, e-mail: qpan@irisa.fr
[4]INRIA, e-mail: christophe.macabiau@inria.fr
[5]INRIA, e-mail: Yves.Sorel@inria.fr

Contents

Foreword

SIGNAL is a language based on synchronized data flow (flows + synchronization) domain: a process is a set of equations on elementary flows describing both data and control. The privileged application context is that of embedded applications. The SIGNAL language has been designed at INRIA-Rennes.

In its INRIA version, SIGNAL is provided with an interactive design toolset called Polychrony. It is composed of a graphical editor and a set of compiling functionalities viewed as a set of services that can be applied or not according to the objectives (simulation, formal verification, optimizations, separate compilation, generation of distributed code, temporal profiling, etc.). The functionalities can be applied either from the graphical editor, or independently.

There is also a commercial version of SIGNAL, which is provided by the TNI-Valiosys company through the Sildex environment[1].

SYNDEX is a system level CAD software for rapid prototyping and optimizing the implementation of real-time embedded applications on multicomponent architectures. It has been designed at INRIA-Rocquencourt.

The main objectives of the *Signal2SynDEx* translation are the following:

- to give access to the functionalities of SYNDEX to SIGNAL designs, in particular the possibility to get and to prototype distributed implementations obtained from quantitative criteria;
- to provide SIGNAL as a possible input formalism for SYNDEX users;
- to allow the use of SYNDEX on applications developed in formalisms for which it is of interest to have a SIGNAL intermediate representation.

The translation is described from the internal abstract representation of a SIGNAL program (as it is implemented in the INRIA version) to the input format of SYNDEX.

The reference document for SIGNAL is [1]. See for example [2] for an overview. Descriptions of the abstract representation may be found in [3, 4]. More documentation can be found on the following site:
http://www.irisa.fr/espresso/espresso-english.html

The SYNDEX CAD Software (see for example [5] for an overview) and its Web Documentation are available on the following site:
http://www-rocq.inria.fr/syndex

[1] http://www.tni.fr/

Chapter 1

Specification of the SIGNAL $\rightarrow$ SynDEx translation

The SIGNAL $\rightarrow$ SYNDEX translator is denoted *SiSy*.

1.1 The SIGNAL hierarchical dependence graph

The abstract representation of a SIGNAL program is a structure called *Hierarchical Conditional Dependence Graph* (HCDG), which is a generalization of Directed Acyclic Graphs.

A HCDG may be defined as follows:

$< G, C, \Sigma, f_N, f_\Gamma, \delta_\Gamma >$ is a Hierarchical Conditional Dependence Graph iff:

- $G =< N, \Gamma, I, O >$ is a dependence graph (DAG) $< N, \Gamma >$ with communication nodes, inputs I and outputs O such that $I \subseteq N$, $O \subseteq N$ and $I \cap O = \emptyset$; the set of nodes N is partitioned into N_d, the set of data nodes, and N_c, the set of clock nodes.

- $< C, \Sigma >$ is an equational control representation where Σ is a set of constraints over a set C of characteristic functions called clocks.

- $f_N : N \rightarrow C$ is a mapping labeling each node with a clock; it specifies the existence condition of the nodes.

- $f_\Gamma : \Gamma \rightarrow C$ is a mapping labeling each edge with a clock; it specifies precedence relations between the nodes.

- $\delta_\Gamma : \Gamma \rightarrow C$ is a mapping labeling each edge with a clock; it specifies data dependences between the nodes.

If $f_N(n) = \hat{c}$, there exists a node $c \in N_c$ such that $f_N(c) = \hat{c}$ and $f_\Gamma(c, n) = \hat{c}$. This node is associated with a Boolean signal `c`.

The equational control representation $< C, \Sigma >$ is expressed over a set C of characteristic functions called clocks. Clocks represent sets of instants. For any node x associated with a *signal* `x`, its clock $\hat{x}$ is equal to 1 if and only if `x` is carrying a data at the considered instant. The Boolean signal `c` that represents the clock of a given signal `x` is such that at every instant t, `c` holds the value *true* if and only if `x` is present.

The equation system Σ evolves in a Boolean algebra $\mathcal{B} =< C, \vee, \wedge, \hat{0}, \hat{1} >$ where $\hat{0}$ denotes the least element of $\mathcal{B}$ which stands for the never present clock.

The following properties hold on the HCDG:

- an edge cannot exist if one of its extremity nodes does not exist: $\forall (x, y) \in \Gamma, (f_N(x) = \hat{0} \vee f_N(y) = \hat{0}) \Rightarrow f_\Gamma(x, y) = \hat{0}$
- the HCDG is deadlock free, which is expressed by: for every cycle $x_1, \ldots, x_n, x_1$ in G, $f_\Gamma(x_1, x_2) \wedge \ldots \wedge f_\Gamma(x_{n-1}, x_n) \wedge f_\Gamma(x_n, x_1) = \hat{0}$.

The equational control representation $< C, \Sigma >$ defines a clock system s organized as a *clock hierarchy.* Such a clock system is a function $s : C \rightarrow C \cup \{tick\}$ such that for every x in C, there is a single clock r, and a minimal integer n such that $s(s^n(x)) = s^n(x) = r$[1]; r is called a root.

The clock hierarchy is a hierarchical tree-like representation of the control of a program: it is represented as a *forest of clocks.* A tree of the forest can be viewed as a representation of an inclusion relation between clocks: in a clock tree, a node is included in its parent (more exactly, the set of instants represented by a node of the clock tree is included in the set of instants represented by its parent). More generally, a node is included in its ancestors.

As we said, each clock of the clock hierarchy is represented by a Boolean signal. In SIGNAL, a special sub-type of the Boolean type, called *event* type, is distinguished: the single value of *event* type is the Boolean value *true.* In this way, clocks can be represented as signals of *event* type.

For a given signal, several Boolean signals can represent its clock: such Boolean signals have the value *true* if and only if the clock is present. One of them, which never holds *false,* is canonical: it is represented by an *event.*

For a given program, different levels of representation of its clock hierarchy are distinguished:

The DC+ level is such that clocks are represented as signals of *event* type.

The bDC+ level (Boolean DC+) is such that clocks are represented as Boolean signals (no *event* type is used). Note that Boolean signals representing clocks have themselves clocks represented as Boolean signals: the clock hierarchy still exists in bDC+.

However, the clock hierarchy in bDC+ may be completed and is provided with a single root, which is *tick.*

The STS level (Symbolic Transition System)—also called "flat bDC+"—is a bDC+ level in which the hierarchy of clocks is *flat:* for every Boolean clock signal b, $s(b)$=*tick.*

Moreover, each *state variable* (corresponding to delayed signals) is defined at *tick.*

[1] $s^1(x)$ is $s(x)$ and $s^{n+1}(x)$ is $s^n(s(x))$

These different levels correspond to different phases of the compilation of a SIGNAL program. They can be more or less appropriate as intermediate representations used to generate input formats for other tools. For example, bDC+ is used as input for code generation, STS is used for some provers. Functionalities exist, that go from DC+ to bDC+ and from bDC+ to STS.

For generating SYNDEX code with the current version of SYNDEX, STS is the appropriate level since SYNDEX considers remanent variables (in STS, all the clocks, which are the variables representing the control of the application, have been expanded upto the most frequent clock, and the state variables are also defined at the most frequent clocks: other variables can have a *don't care* interpretation at the instants at which they are "absent").

Syntactic representation of the HCDG

There exists a syntactic representation of the HCDG in the proper syntax of SIGNAL (this compiled form of a program `P` appears in a file called `P_TRA.SIG` for the DC+ level, `P_bDC_TRA.SIG` for the bDC+ level). For this reason, we will use the SIGNAL syntax to describe source of the *SiSy* translation.

1.2 Implementation graph

The explicit representation of the HCDG is an implementation graph such that clocks are represented as signals and clock expressions are represented as expressions on these signals. Moreover, each signal is associated with a node that produces it.

A graph P (corresponding to a *process model*—or shortly, a *process* `P`) is described in particular by the following elements:

- its name `P`;
- its signals with their type, among which are distinguished the signals representing clocks;
  within the set of signals, the following ones are distinguished:
  - its inputs $\mathtt{i}_1, \ldots, \mathtt{i}_m$, the types of which are respectively $\mathtt{t}_1, \ldots, \mathtt{t}_m$,
  - its outputs $\mathtt{o}_1, \ldots, \mathtt{o}_n$, the types of which are respectively $\mathtt{u}_1, \ldots, \mathtt{u}_n$;
- its clock hierarchy;
- its nodes $p_1, \ldots, p_q$;
- its data dependences $d_1, \ldots, d_r$: each one is expressed by a data dependence `u -> v when c`, where `u`, `v` are nodes of the graph, and `c` is a clock;
- its precedence relations $c_1, \ldots, c_s$: each one is expressed by a dependence `u -> v when c`, where `u`, `v` are nodes of the graph, and `c` is a clock;

- a function *node_of* associates with each signal the node that defines it;
- a function *sg* associates with each clock the set of the nodes which are defined at this clock (a given node is associated with a single clock by the function *sg*):
  Let us denote by $\hat{x}$ the clock of a given signal x, if $node\_of(x) = p$ then $p \in sg(\hat{x})$.

Data dependences are set between the nodes of the graph according to the rules associated with operators. Precedence relations result from explicit scheduling expressed in the program or from added dependences.

Data flow connections appear in the graph through the references to signals appearing in the expressions that define the nodes.

The different sorts of nodes are the following:

- Equation,
- Memorization,
- Partial definitions,
- Sub-graph,
- External call,
- Process call,
- Array of processes.

Each node has a set of inputs and a set of outputs.

There is a data flow connection from the i^{th} output of a node p_k to the j^{th} input of a node p_l if a given signal a is produced as i^{th} output of p_k and used as j^{th} input of p_l.

Refer to [1] for a formal description of the *processes* corresponding to these nodes, the definition of their clocks, etc.

1.2.1 Equation

An equation is a node `X := E` associated with the signal `X`, where `E` is an expression with the following form:
$(\mathtt{E}_1$ `when` $\mathtt{c}_1)$ `default ...` $(\mathtt{E}_n$ `when` $\mathtt{c}_n)$.
The expressions $\mathtt{E}_i$ are synchronous expressions containing only *static monochronous operators* (corresponding to standard operators extended to sequences). The $\mathtt{c}_i$ represent clocks which are mutually exclusive.

1.2.2 Memorization

A memorization is a node `ZX := X $m window n` associated with the signal `X`.
Note that for `ZX := X $m`, no data dependence in induced from this equation from the node associated with `X` to the node associated with `ZX`.

1.2.3 Partial definitions

A list of partial definitions `X ::= E`$_1$, ..., `X ::= E`$_p$ is a node associated with a signal `X`. Each `E`$_i$ is an expression which has the same form as the right-hand side expression of an equation `X := E`. Each partial definition is a node analogous to the node representing an equation.

1.2.4 External call

An external call is a node `(s`$_1$`, ..., s`$_q$`) := Q(e`$_1$`, ..., e`$_p$`)` where `Q` designates an external process with p inputs and q outputs.

Particular external calls are named `read` and `write`.

In the SIGNAL graph, additional `read` and `write` nodes are associated respectively with the inputs and outputs of the graph.

Although they are distinguished in the implementation graph, external calls are just special cases of process calls.

1.2.5 Process call

A process call is a node `(s`$_1$`, ..., s`$_q$`) := Q(e`$_1$`, ..., e`$_p$`)` where `Q` designates a process with p inputs and q outputs. This node contains a reference on the graph associated with `Q` (representing a *process model* which can be that of a *process,* an *action,* a *node* or a *function*—cf. [1]).

In the construction of the graph of a program containing a process call, an additional `read` external call is associated with each output of the process call.

1.2.6 Sub-graph

A sub-graph is a node `e :: (| Q |)` associated with a label `e` (`Q` is a process expression). The nodes of a sub-graph are a subset of the nodes of the graph that contains it. A sub-graph has no proper clock hierarchy associated with it.

1.2.7 Array of processes

An array of processes is a node `array i to n of Q end` where `Q` designates a process expression represented as a sub-graph.

1.3 Translation of SIGNAL predefined types and operators

SYNDEX predefined libraries are defined, that contain the declarations of all *algorithms* corresponding to predefined operators in SIGNAL. The considered SIGNAL predefined operators are all the basic *static monochronous operators* (corresponding, as we said, to standard operators extended to sequences), plus the *dynamic monochronous operators* which are delay and window. The other basic SIGNAL predefined operators (`when`, `default` and operators derived from them) do not have corresponding *algorithms* in SYNDEX predefined libraries: they are translated in a specific way that will be explained in the following.

One single predefined library, `signal.sdx`, could contain the declarations of all *algorithms* corresponding to predefined operators in SIGNAL. However, it may be preferable to structure the libraries according to the different types. In that case, a library `T.sdx` contains the declarations of *algorithms* associated with the type (or domain of values) `T`.

Remarks:

- A given operator that manipulates objects of different types is associated with one of these types and declared in the corresponding library (example: relation operators).
- For a given polymorphic SIGNAL operator (example: delay operator), there exists as much corresponding SYNDEX *algorithms* as there are possible combinations of types for it.

Let us consider some SIGNAL operator named F defined with the following signature: $t_1 \times \ldots \times t_n \rightarrow t_{n+1} \times \ldots \times t_{n+p}$

The *SiSy* generic translation of this operator is an *algorithm* named $SiSy(F)$ declared with:

- n inputs denoted $i_1, \ldots, i_n$, the types of which are respectively $SiSy(t_1), \ldots, SiSy(t_n)$,
- one added input, named *precedence_in*, the type of which is *void*,
- p outputs denoted $o_1, \ldots, o_p$, the types of which are respectively $SiSy(t_{n+1}), \ldots, SiSy(t_{n+p})$,
- one added output, named *precedence_out*, the type of which is *void*.

The *precedence_in* and *precedence_out* added ports serve to specify precedence dependences in SYNDEX (dependence without data exchange that states that an operation has to be executed before another one).

The *SiSy* translation of a SIGNAL static monochronous operator is what is called an *operation* in SYNDEX (appears with the keywords `def algorithm` in SYNDEX). The *SiSy* translation of a SIGNAL dynamic monochronous operator is a *memory* in SYNDEX (appears with the keywords `def memory`).

Examples

- Let us consider the following SIGNAL operator:
  + : *integer* $\times$ *integer* $\rightarrow$ *integer*
  The following SYNDEX *algorithm* will be associated with it in the `integer.sdx` library:

```
def algorithm add:
 ? void precedence_in;
 ! void precedence_out;
 ? integer i_1;
 ? integer i_2;
 ! integer o_1;
```

- The SIGNAL delay operator applying on the *integer* type (example of use of this operator in SIGNAL: `zx := x $ init v`) is translated by the following SYNDEX memory declared in the `integer.sdx` library:

```
def memory delay [0..0]:
 ? void precedence_in;
 ! void precedence_out;
 ? integer init;
 ? integer i_1;
 ! integer o_1;
```

To get a better readability, the input corresponding to the initialization value has the special name `init`. The *range* (here `[0..0]`) corresponds to memorized values (here, one value has to be memorized, which is the value taken at the previous instant: if t is the current instant, this previous instant is $t-(0+1)$).

For the parameterized delay `zx := x $n init v`, the following memory can be declared:

```
def memory n_delay <n> [n-1..n-1]:
 ? void precedence_in;
 ! void precedence_out;
 ? integer[n] init;
 ? integer i_1;
 ! integer o_1;
```

For a sliding window `zx := x window n init v`, the following memory can be declared:

```
def memory window <n> [0..n-2]:
 ? void precedence_in;
 ! void precedence_out;
 ? integer[n-1] init;
 ? integer i_1;
 ! integer[n] o_1;
```

For a combination of sliding window and delay `zx := x $m window n init v`, the following memory can be declared:

```
def memory delayed_window <m,n> [m-1..m+n-2]:
 ? void precedence_in;
 ! void precedence_out;
 ? integer[m+n-1] init;
 ? integer i_1;
 ! integer[n] o_1;
```

The SynDEx library associated with a type `T` contains the declaration of *constants* (used to represent constant vertices), that deliver constant values of that type. Such a *constant* is declared as a `def constant` with parameters the value of which are used to represent the value of the constant (the *constant* has one output, which delivers this value).

The SynDEx library associated with a type `T` contains also the declaration of *sensors* and *actuators* allowing to read and write elements of type `T`:

```
def sensor input:
 ? void precedence_in;
 ! void precedence_out;
 ! T o_1;

def actuator output:
 ? void precedence_in;
 ! void precedence_out;
 ? T i_1;
```

1.4 Non elementary types

In the same way that it is the case for predefined types and associated operators, each non elementary or external type used in a program must be characterized by a given name in the *SiSy* translation of the program, and all the operators that may apply to it must be declared as *algorithms.*

The only difference with predefined types is that these declarations cannot exist *before* the translation of the program that contains these types.

Examples of such types are enumerated types, structures, and external types.

Arrays, which are also non elementary types, are distinguished since SynDEx provides an array constructor (one dimension arrays). This constructor is used to represent the array types of the program. However, all the operators on arrays (including for example access to an element) must be explicitly declared as *algorithms*.

1.5 Translation of a graph

We will use the notations defined in section 1.2.

1.5.1 Interface

The *SiSy* translation of a graph named `P` (corresponding to a *process model* named `P`) is an *algorithm* named `P`:
`def algorithm P:`.

Parameters of the algorithm in SynDEx correspond directly to the Signal parameters of the *process model*.

For the *SiSy* translation, the signals of the interface of a graph are translated differently according to the following two cases:

1. the input or output signal is considered as communicating with the external world (upper level communication),

2. the input or output signal stands for a communication with another graph.

In the *SiSy* translation, it is considered that all the interface signals of a given Signal graph (as it is before the possible transformations of some sub-graphs into graphs, which are done in particular to translate the clock hierarchy) have the same status with respect to this distinction.

If P is a graph the interface of which represents upper level communications, the *algorithm* `P` is declared with no input and no output. Instead, the description of `P` contains references to sensors and actuators corresponding to each one of the inputs (sensors) and outputs (actuators) of P.
If `tt_i` generically designates the SynDEx library associated with the type $\mathtt{tt}_i$, the *algorithm* `P` contains the following references associated with the inputs and outputs of the graph P:

```
t_1/input[1] i_1;
...
t_m/input[1] i_m;
u_1/output[1] o_1;
...
u_n/output[1] o_n;
```

Remark: in SynDEx, the *dimension* `[1]` expresses that there is *one* instance of the corresponding vertex (apart from arrays of processes, this is the general case we have to consider).

Due to the translation of the clock hierarchy (cf. section 1.5.4), the different sensors and actuators do not necessarily appear in the same *algorithm.*

If P is a graph the interface of which represents communications with other graphs, the *algorithm* `P` is declared with the inputs and outputs corresponding to those of the graph, plus *precedence_ in* and *precedence_ out* added ports:

```
? void precedence_in;
! void precedence_out;
? t_1 i_1;
...
? t_m i_m;
! u_1 o_1;
...
! u_n o_n;
```

1.5.2 Nodes

The nodes of the graph are translated by references to vertices in the description part of the *algorithm* associated with the graph.

1.5.2.1 Constant

The constants are explicitly represented by references to *constant* vertices in SynDEx (associated dependences follow trivially).

1.5.2.2 Equation

Remind that the general form of an equation is `X := (`$\texttt{E}_1$` when `$\texttt{c}_1$`) default ... default (`$\texttt{E}_n$` when `$\texttt{c}_n$`)`.

However, we will see that it is sufficient to consider equations such as `X :=` $\texttt{E}_i$ `when` $\texttt{c}_i$, or, more exactly, partial definitions of `X` such as `X ::=` $\texttt{E}_i$ `when` $\texttt{c}_i$ (in general, there are several such partial definitions for `X`).

In `X ::=` $\texttt{E}_i$ `when` $\texttt{c}_i$, $\texttt{c}_i$ represents the clock of the considered definition of `X`. We will see below how clocks are handled in the *SiSy* translation. So we have only to consider here equations such as `X ::=` $\texttt{E}_i$ with $\texttt{E}_i$ a synchronous expression.

Moreover, it is obvious that it suffices to consider elementary expressions (using only one operator): adding auxiliary variables, it is always possible to rewrite a definition containing composed expressions into a composition of elementary definitions.

Thus we consider a definition `X ::=` F`(`$\texttt{Y}_1$`, ..., `$\texttt{Y}_n$`)` where F is some Signal n-ary operator, the translation of which is an *algorithm* named `F` (for example) declared in some library `T`. The *SiSy* translation of this definition is a reference to the *algorithm* `F` associated with F. This reference must have a unique name in the *algorithm* that contains it, for example, this name can be the name of the *algorithm,* `F`, suffixed with the current value of some counter of references:

```
T/F[1] F_k
```

To specify the dependences, this occurrence is designated by its name `F_k`.

Remark: just like above, the *dimension* `[1]` expresses that there is *one* instance of the corresponding vertex.

1.5.2.3 Memorization

The *SiSy* translation of a memorization `ZX := X $m window n` is analogous to the translation of a definition, except that it is a reference to a *memory.* The considered sort of *memory* (for example, `delay`, `n_delay`, `window`, `delayed_window`) depends on the values of `m` and `n`.

Just like other references, this reference must have a unique name in the *algorithm* that contains it.

1.5.2.4 Partial definitions

For the *SiSy* translation of partial definitions, the original graph is considered as if it was rewritten without the nodes representing the *list* of partial definitions associated with a given signal `X`. Then, the signal `X` is considered as being produced by all the nodes representing its partial definitions, and the connections or dependences involving `X` are transfered consequently on these nodes.

With respect to this transformation (which may be virtual), the translation of a partial definition is exactly that described in section 1.5.2.2.

1.5.2.5 External call

The *SiSy* translation of an *external process* Q is just like the *SiSy* translation of any *process,* except that the description part of the corresponding *algorithm* is empty since Q is not defined in SIGNAL: *SiSy*(Q) is an *algorithm* named `Q` with an explicit interface corresponding to the interface of Q (this is analogous to the translation of SIGNAL basic operators).

Then, the *SiSy* translation of an external call of the process Q is a reference to the algorithm `Q` (with a unique name in the *algorithm* that contains it, just like other references):

```
Q[1] Q_k
```

The additional `read` and `write` nodes associated with the inputs and outputs of a SIGNAL graph are not translated explicitly as references to corresponding algorithms in SYNDEX: instead, the considered inputs and outputs appear as explicit inputs and outputs in SYNDEX and the concerned data flow connections and dependences apply on these inputs and outputs (or on the corresponding references to sensors and actuators): cf. the description of the translation of the interface of a graph in section 1.5.1.

In the same way, the additional `read` nodes associated with the outputs of a process call are not translated explicitly in SYNDEX: cf. below the translation of a process call.

The concerned data flow connections and dependences apply directly on the outputs of the referred *algorithm* corresponding to the called process.

1.5.2.6 Process call

The *SiSy* translation of a *process* (or sub-process) Q is precisely the object of all section 1.5: $SiSy(Q)$ is an *algorithm* named `Q`.

Then, the *SiSy* translation of a call to the process Q is a reference to the algorithm `Q` (with a unique name in the *algorithm* that contains it, just like other references):
`Q[1] Q_k`

1.5.2.7 Sub-graph

The nodes and dependences of a sub-graph are subsets of the nodes and dependences of the graph that contains it. So, they are handled in the usual way by the *SiSy* translation. In this sense, a sub-graph has no specific translation: its *SiSy* translation is just the *SiSy* translation of the nodes and dependences that it contains.

If, for some reason, a sub-graph has to be kept as a proper object, it has first to be transformed into a graph and replaced into its including graph by a process call to this new graph.

This is a functionality available in the SIGNAL environment.

Then, the *SiSy* translation is the same as for any process call.

1.5.2.8 Array of processes

to be described

1.5.3 Dependences

The edges of the SYNDEX graph are built from:

- the data flow connections of the SIGNAL graph (represent the *SiSy* translation of the composition operator of SIGNAL),
- the explicit precedence relations.

They do not come from the data dependences that have been set in the graph.

Data flow connections

With each data flow connection d from the i^{th} output of a node p_k to the j^{th} input of a node p_l, the *SiSy* translation associates a *strong precedence data* from the output of the vertex $SiSy(p_k)$ corresponding to the i^{th} output of p_k, to the input of the vertex $SiSy(p_l)$ corresponding to the j^{th} input of p_l.

In SYNDEX, a reference to some vertex $SiSy(p_r)$ in a given *algorithm* has a unique name: let us call $\texttt{p\_k}_s$ and $\texttt{p\_l}_t$ the unique names of the considered references to the vertices $SiSy(p_k)$ and $SiSy(p_l)$. If `i_j` designates the j^{th} input (non including the *precedence_in* one) of the vertex $SiSy(p_l)$, and `o_i` designates the i^{th} output (non including the *precedence_out* one) of the vertex $SiSy(p_k)$, the *strong precedence data* associated with d is:

`strong_precedence_data` $\texttt{p\_k}_s\texttt{.o\_i -> p\_l}_t\texttt{.i\_j;}$

However, when the *SiSy* translation of the graph keeps explicitly the declaration of the inputs and outputs of the graph (when they are not represented as sensors or actuators), particular cases have to be distinguished when one of the extremities of a data flow connection is such an input or output.

In this case, if the i^{th} output of the node p_k (which may be a `read`) is also the $i'^{\ th}$ input of the graph, then the *strong precedence data* is from the corresponding input of the *SiSy* translation of the graph, to the input of the vertex $SiSy(p_l)$ corresponding to the j^{th} input of p_l.

In the same way, if the j^{th} input of the node p_l (which may be a `write`) is also the $j'^{\ th}$ output of the graph, then the *strong precedence data* is from the output of the vertex $SiSy(p_k)$ corresponding to the i^{th} output of p_k, to the corresponding output of the *SiSy* translation of the graph.

Example

Consider the following SIGNAL process:

```
process P =
  ( ? integer a, b, c;
    ! integer x, y; )
  (| x := a + b
   | y := x + c
   |)
;
```

Supposing the translation of this process keeps its interface, a corresponding SYNDEX *algorithm* is:

```
def algorithm P:
 ? void precedence_in;
 ! void precedence_out;
 ? integer a;
 ? integer b;
 ? integer c;
 ! integer x;
 ! integer y;
conditions: true;
references:
 integer/add[1] add_1;
 integer/add[1] add_2;
```

```
dependences:
 strong_precedence_data a -> add_1.i_1;
 strong_precedence_data b -> add_1.i_2;
 strong_precedence_data add_1.o_1 -> x;
 strong_precedence_data c -> add_2.i_1;
 strong_precedence_data add_1.o_1 -> add_2.i_2;
 strong_precedence_data add_2.o_1 -> y;
```

Precedence relations

With each precedence relation in the graph, the *SiSy* translation associates a *precedence* defined as follows.

Let `u -> v when c` such a precedence relation where `u`, `v` are nodes of the graph (and `c` is a clock). The *SiSy* translation associates with it a *precedence* from the *precedence_ out* port of the vertex $SiSy(u)$ to the *precedence_ in* port of the vertex $SiSy(v)$ (the clock `c` is not taken into account in this translation).

When the *SiSy* translation of the graph keeps explicitly the declaration of the inputs and outputs of the graph (when they are not represented as sensors or actuators), particular cases have also to be distinguished when one of the extremities of a precedence relation is associated with such an input or output.

In this case, if `u` or `v` are associated respectively with the input i_k or the input i_l of the graph, then the corresponding extremity of the *precedence* is the considered input of the *SiSy* translation of the graph.

In the same way, if `u` or `v` are associated respectively with the output o_k or the output o_l of the graph, then the corresponding extremity of the *precedence* is the considered output of the *SiSy* translation of the graph.

1.5.4 Clock hierarchy

Since for the *SiSy* translation the clock hierarchy is a STS one, the clock tree contains at most two levels:

- the level of the most frequent clock, *tick*, which is the root of the tree,
- the level of its sons, which are less frequent clocks defined by the *true* values (and symmetrically, the *false* values) of Booleans. Let us denote $\mathtt{b}_1, \ldots, \mathtt{b}_m$ these Booleans. Each $\mathtt{b}_i$ represents two clocks: the clock `when` $\mathtt{b}_i$ (which denotes the instants at which $\mathtt{b}_i$ is *true*) and the clock `when (not` $\mathtt{b}_i$`)` (which denotes the instants at which $\mathtt{b}_i$ is *false*). Each one of these clocks is a son of the clock *tick*.

With each clock of the clock hierarchy, it is associated (by the function *sg*) the sub-graph containing the nodes which are defined at this clock. In particular, the nodes corresponding to the definitions of the Booleans $\mathtt{b}_1, \ldots, \mathtt{b}_m$ are associated with *tick*.

Note that in general, there are dependences from the nodes of a given sub-graph to nodes of other sub-graphs.

Among the different sorts of nodes, some represent different expressions of definition of signals, which apply according to different clocks. This is the case of the general form of an equation:

`X := (E`$_1$ `when c`$_1$`) default ... default (E`$_n$ `when c`$_n$`)`

Such a definition is associated with the Boolean clock representing the clock of `X`. But it is composed of n partial definitions of `X` which are the following ones:

`X ::= E`$_1$ `when c`$_1$

...

`X ::= E`$_n$ `when c`$_n$

The `c`$_i$'s are exclusive clocks, and the clock `c`$_j$ is the clock at which `X` is defined by the synchronous expression `E`$_j$.

Thus, it is always possible to *split* an equation

`X := (E`$_1$ `when c`$_1$`) default ... default (E`$_n$ `when c`$_n$`)`

into n partial definitions.

Each partial definition `X ::= E`$_j$ `when c`$_j$ is associated with the clock `c`$_j$ in the graph. Then, at this clock, the considered partial definition of `X` is `X ::= E`$_j$.

Within the transformation, dependences are changed accordingly.

This transformation is a functionality which has to be made available in the SIGNAL environment.

Note that the same transformation holds for the partial definitions of the initial graph which are expressed as compositions of `default`.

We consider that this transformation has been applied for the *SiSy* translation of a graph P.

Then, the sub-graphs associated with each one of the clocks which is a son of *tick* are transformed into graphs (cf. section 1.5.2.7). For each `b`$_i$, the graph $g+_i$ associated with the clock `when b`$_i$ and the graph $g-_i$ associated with the clock `when (not b`$_i$`)` are grouped together into a single graph g_i, composed of two sub-graphs: $sg+_i$ (corresponding to the sub-graph associated with the clock `when b`$_i$) and $sg-_i$ (corresponding to the sub-graph associated with the clock `when (not b`$_i$`)`).

Each graph g_i is then translated following the usual *SiSy* translation of graphs, in which, however, a *conditioning* with two branches is defined, corresponding to the two possible values of the Boolean `b`$_i$ (which is an input of g_i): *true* and *false* (note that it has to be considered as an integer, using for example values 1 and 0, in SYNDEX).

Let `b_i` designate the integer corresponding to `b`$_i$. The two branches of the conditioning, `b_i = 1` and `b_i = 0`, correspond respectively to the translations of the sub-graphs $sg+_i$ and $sg-_i$.

The translation of g_i can be sketched as follows:

```
def algorithm P_b_i:
?  ...
!  ...
conditions:  b_i = 1
SiSy(sg+i)
conditions:  b_i = 0
SiSy(sg-i)
```

Note that another possible translation would be to have one graph associated with each clock which is a son of *tick*, then each graph $g+_i$ is translated as an *algorithm* which contains a *conditioning* with a single branch `b_i = 1` and each graph $g-_i$ is translated as an *algorithm* which contains a *conditioning* with a single branch `b_i = 0`. Such a translation would allow to have precedences between $SiSy(g+_i)$ and $SiSy(g-_i)$.

The rewritten graph P contains the nodes associated with the clock *tick*, plus process calls corresponding to each one of the graphs associated with the sub-clocks of *tick*. The *SiSy* translation of P is an *algorithm* `P` obtained in the usual way (in particular, it contains the translations of all the nodes of P, those associated with *tick*, and those representing these process calls). The definitions of this *algorithm are not* conditioned (in SynDEx: `conditions: true;`).

Note that for a given signal `X` which is partially defined at different sub-clocks of *tick*, there are corresponding definitions in each one of the algorithms associated with these sub-clocks, the results of which are provided as outputs of these algorithms. Then, since all these definitions contribute to the overall definition of `X`, there are dependences from each one of these outputs onto the vertices corresponding to a use of `X` in the *algorithm* `P` (or onto the output of `P` corresponding to `X`).

Chapter 2

Implementation

to be described

Bibliography

[1] L. Besnard, Th. Gautier, P. Le Guernic, *SIGNAL V4 – INRIA version: Reference Manual,* Draft Report, IRISA, February 2002.

[2] P. Le Guernic, Th. Gautier, M. Le Borgne, C. Le Maire, "Programming Real-Time Applications with Signal", *Proceedings of the IEEE,, 79,* 9, p. 1321–1336, September 1991, ftp://ftp.irisa.fr/local/signal/publis/articles/ProcIEEE-91:gen_lang.ps.

[3] T. P. Amagbegnon, L. Besnard, P. Le Guernic, "Implementation of the Data-flow Synchronous Language SIGNAL", *Proceedings of the ACM Symposium on Programming Languages Design and Implementation (PLDI'95),* ACM, p. 163–173, 1995, ftp://ftp.irisa.fr/local/signal/publis/articles/PLDI-95:compil.ps.gz.

[4] P. Aubry, P. Le Guernic, S. Machard, "Synchronous distribution of SIGNAL programs", *Proc. of the 29th Hawaii International Conference on System Sciences,* IEEE Computer Society Press, p. 656–665, January 1996, ftp://ftp.irisa.fr/local/signal/publis/articles/HICSS-96:distr.ps.gz.

[5] Th. Grandpierre, Ch. Lavarenne, Y. Sorel, "Optimized Rapid Prototyping For Real Time Embedded Heterogeneous Multiprocessors", *CODES'99 7th International Workshop on Hardware/Software Co-Design,* Rome, May 1999.