SIGNAL to SYNDEX: Translation Between Synchronous Formalisms*

Qiuling PAN[1], Thierry GAUTIER[1], Loïc BESNARD[2], and Yves SOREL[3]
{*qpan, gautier, lbesnard*}*@irisa.fr, Yves.Sorel@inria.fr*

[1] IRISA / INRIA
[2] IRISA / CNRS
F-35042 RENNES, France

[3] INRIA, Domaine de Volucean-Rocquencourt BP 105
78153 Le Chesnay Cedex

Abstract. This paper presents techniques used for the translation between two synchronous languages: SIGNAL and SYNDEX (algorithm model). Both are synchronous formalisms and their corresponding integrated environments can be used for synchronous executions including GUI editing, compilation, simulation, testing, optimization, and code generation. While the two powerful environments have their inherent features and advantages, a translation path from SIGNAL to SYNDEX makes it possible to access SYNDEX functionalities from SIGNAL programs.

Keywords. synchronous, SIGNAL, SYNDEX, translation, real-time applications, optimized distribution

1 Introduction

The fundamental theory of the synchronous approach is discrete event systems and automata theory. Time is handled according to the first two of its three following characteristic aspects: partial order of events, simultaneity of events, and finally delays between events. So real-time systems are designed assuming that time is just a succession of events and that computation is performed instantaneously, with no duration, although this idea is unusual in the software domain. For the advantages of this simplification, please refer to [ABL95]. SIGNAL is a programming language particularly suited for real-time applications, reactive, and embedded systems. It is a synchronous language, as opposed to the asynchronous approach, e.g. Ada, C. Like LUSTRE [H91], its style is declarative, to be compared with imperative synchronous style in ESTEREL [BD91]. SYNDEX (Synchronized Distributed Executive) environment is a graphical interactive software implementing the AAA methodology [S03]. AAA means Algorithm, Architecture and Adequation(French word for efficient matching). In

* This work is supported by the RNTL (Réseau National de recherche et d'innovation en Technologies Logicielles) project ACOTRIS [A03].

this methodology, the algorithm model has the semantics of synchronous languages. At this point, it is assumed that the result produced by an operation is simultaneous with the input by which it is triggered. SIGNAL and SYNDEX environments are both very useful applications for real-time systems. Being independent, each has its own features. SIGNAL is a language defined via a formal mathematical model, which gives it the inherent power in validation and verification of programs. SYNDEX on the other hand has its dedicated architecture model with predefined hardware components that provides it with an easy and efficient way to describe distributed architectures (microprocessors along with specific integrated circuits, connected by a network). Further on, the optimized executive codes considering minimizing hardware resources while satisfying real-time and technological constraints can be generated by running "adequation" of SYNDEX. This paper describes the development of an interface from the SIGNAL language to the SYNDEX algorithm model. It is denoted by *SiSy*.

The rest of this paper is organized as follows. The next two sections present brief summaries of the source (SIGNAL) and the target (SYNDEX) language. Then follows an explanation of the translation strategy. It describes the theory of the underlying approach, gives the translation rules we have determined along with some examples. Concluding remarks are presented in the final section.

2 The synchronous language SIGNAL

In the SIGNAL language, a signal is a sequence of values of the same type, which are present at some instants. The set of instants where a signal is present is the clock of the signal. At a given instant (denoted by t), signals may have the status present, or absent – the latter is denoted by the special symbol $\perp$ in the semantic notation. Only when a signal is present it can carry a value. Signals that are present simultaneously for any environment are said to have the same clock, otherwise they have different clocks. The syntax `^x` is used to denote the clock

2. **Reference to past values:** delay operator $ gives access to past values of a signal. Delay is also a *monochronous* operator.
   In SIGNAL simple delay is: `y := x $ init i0` which specifies $x_t \neq \perp \Leftrightarrow y_t \neq \perp,\ \forall t > 0$: $y_t = x_{t-1}, y_0 = i_0$.
3. **Downsampling:** extracts values from a signal with the action of a boolean expression. It produces a resulting signal with a less frequent clock and is therefore counted among the *polychronous* operators.
   In SIGNAL: `y := x when c` means: $y_t = x_t$ if $c_t = true$, else $y_t = \perp$.
4. **Deterministic merge:** merges two signals while putting a priority on the first. The clock of the result is the union of the two operands. Since the merge handles signals with different clocks, it is also a *polychronous* operator.
   In SIGNAL: `y := x default z` means: $y_t = x_t$ if $x_t \neq \perp$, else $y_t = z_t$.
5. **Composition of processes:** the above operations may be composed with the commutative and associative operator `"|"` like `" P | Q"` which defines the greatest relation constraining their common signals to respect the constraints imposed respectively by P and Q. In fact, this operator is the union of the two equation systems represented by P and Q.

2.2 Extended Language

For convenience, the SIGNAL language offers also many derived operators which are proved to be very useful in most of the applications we developed, some of them are:

- **Synchronizer:** x_1 `^=` x_2 `^=` ... `^=`x_n specifies the equality of the clocks of its operands, i.e., `^x1` = `^x2 = ... = ^xn`.
- **Delay and Window:** `Y:= x $ m window n init E0` is called generalized delay and sliding window, it follows the definition: $x_t \neq \perp \Leftrightarrow y_t \neq \perp,\ \forall (t > 0,\ 0 \leq i < n)$,when $t + i - m \geq n, Y_{t[i]} = y_{t-n+i+1-m}$; when $1 \leq t + i - m < n, Y_{t[i]} = E_{0[t-n+i+2-m]}$. It produces a vector signal Y of length n and m-delay on x.
- **Cell:** `y := x cell b init i0` means that y takes the value of x when x is present, otherwise when b is true, y takes its own latest value. The clock of y is the upper bound of the clock of x and the clock defined by the instants at which b is true. It is obvious this is a *polychronous* operation, it can be expressed like this:
  `y := x default (y $ init i0) | y ^= ^x default (when b)`
- **Partial definitions:** Partial definitions are series of equations used to define a signal. Each one of the partial definition contributes to the overall definition of this signal. These partial definitions can appear in different syntactic contexts. The overall definition of the signal is considered as complete in a compilation unit. They are syntactically distinguished by the use of the special symbol ::=.
- **Domains of values:** The types of SIGNAL can be predefined simple types (integer, real, char,...). Among them, there is a special type named *event* which is a pure signal, useful only for its clock. It can only be present or

absent. SIGNAL also provides composite types including array types, enumerated types, and tuple types. Moreover, it is allowed to declare external types and state variables, a typed sequence of elements that are present as frequently as necessary. Default conversions between different types are also supported.
- **Modularity:** The SIGNAL language allows modular programming and external function calls. It can be used to describe internal or external communications, facilitating encapsulation and reusability. According to the level of abstraction, black box and grey box modules are available.

The SIGNAL language has all the features needed for real-time application programming. It has been proved to possess maximum expressive power for synchronization mechanisms [BLSS87]. It must be pointed out that this section only presents the elementary parts of SIGNAL. Some advanced concepts and the strict formal semantics are not mentioned. For a complete and detailed understanding, please refer to [BGL02], or take a look at the web site of the ESPRESSO [E03].

2.3 The Integrated Environment

The SIGNAL environment is an integrated development environment and technology demonstrator, also called POLYCHRONY, consisting of a compiler, a visual editor and simulator, and a model checker. SIGNAL provides the programmer with the nice high-level constructs needed to describe a real-time system in a simple and elegant way. Nevertheless it is a challenge to construct a compiler, able to generate efficient executable code by using a reasonable amount of computing resources. The SIGNAL compilation performs formal calculus on systems of boolean equations which express the synchronizations in the program. Then, by using an arborescent hierarchical representation of the equations, the compiler checks the consistency of the synchronizations and generates executable code (e.g. C or JAVA) [MCL92]. The key feature of the compiler is that it synthesizes the global timing from the programmers specifications. This is different from other compilers (e.g. LUSTRE), which require the global view of the timing for the programmer to write the program. The model checker, called SIGALI, gives the verification of dynamic properties of the programs.

While SIGNAL fills this task quite well, the increasing complexity of the applications and strict real-time constraints often make it a considerable need to use multiprocessor (parallel, distributed) architectures when real-time algorithms are executed. SYNDEX can automatically generate dead-lock free executables for real-time execution of algorithms on distributed architectures. It is based on a static scheduling policy to minimize execution time and program size overheads. In the end, the M4 macro-processor produces source code in the preferred compilable language.

3 AAA and SYNDEX

The SYNDEX environment is a system level CAD tool, supporting the AAA methodology. It aims at rapid prototyping and optimization of real-time em-

bedded applications on multicomponent architectures. It has been designed and developed at INRIA-Rocquencourt. SynDEx is a graphical interactive software offering a seamless environment to help the user from the specification level (functionalities, hardware resources, real-time and embedding constraints of the application) to the implementation level. It allows user to import programs as text files as well as to use a graphic editor.

3.1 The Integrated Environment

In the SynDEx environment, an algorithm is modeled by a conditioned data-flow graph, that is, a folded dependency graph presenting a pattern indefinitely repeated. In the textual formalism, each graph unit is defined as an algorithm. There are four kinds of special algorithms: *constant, sensor, actuator* and *memory*. The grammar of SynDEx codes will be presented in the following subsection. The dependency graph describes dependencies, represented by edges, between operations, represented by vertices. In the SynDEx algorithm model, there are four kinds of dependencies: `strong_precedence_data, weak_precedence_data, precedence` and `data` dependency [M02]. In fact, most dependencies actually represent a data transfer from a producer operation to a consumer operation. This involves a partial order on the execution of the operations, called potential parallelism. Potential means that this parallelism will be exploited only if parallel hardware resources are available. The execution of each graph pattern of the dependency graph is triggered when an input event, coming from the environment, is received by operations without predecessor. Such special operations are called *sensor*s. The output events that are sent to the environment by operations without successor are called *actuator*s. Moreover, a dependency graph may be conditioned, that is, a part of this graph may not be executed. For example, a conditioning vertex, with two inputs and one output, does not produce any data when its conditioning input, which must be of boolean type, is carrying a false value. In this case, by transitivity, all dependent vertices will not be executed. Otherwise, when both inputs are present, but the condition is true, the output takes the value of the other input. Note that when we talk about algorithm model only, the durations of operations and data transfers defined in the architecture model of SynDEx are not taken into account. In those cases, it produces a logical time which corresponds to the time the input / output events are interleaving. The algorithm model of SynDEx has the semantics of synchronous languages and more specifically, has the programming style of a data-flow language which is just what Signal has.

The goal of the AAA methodology is to find an optimized implementation of an application algorithm on an architecture, while satisfying constraints [KS98]. AAA is based on graph models to exhibit both, the potential parallelism of the algorithm, and the available parallelism of the multicomponent. In this methodology, the first A (Algorithm) is the core part to connect with the *SiSy* translation. In fact, every sub-clock of Signal programs is translated into one *algorithm* in SynDEx as we will explain in the following sections.

3.2 The SynDEx Algorithm Model

The grammar of the SynDEx *algorithm* definition is shown below. Here, the terminal symbols are set in bold. Non-terminal symbols are set in italic. Square brackets represent optional elements. Curly brackets represent zero, one, or several repetitions of the enclosed element. Curly brackets with a trailing plus sign represent one or several repetitions of the enclosed element. Because of the space limitation, some unimportant non-terminal symbol definitions are not listed here. For more details please refer to [M02].

algorithm ::= **def algorithm** *NAME [arg_name][dim_window]* :
{*in_port*}{*out_port*}{*conditions references dependencies*}*[description]*
in_port ::= **?** *NAME [dimension] [NAME] [init_port][rank][position]*
out_port ::= **!** *NAME [dimension] [NAME] [rank] [position]*
conditions ::= **conditions:** *[boolean* | *NAME = integer];*
references ::= **references:** {*ref_type [arg_values] NAME [position];*}
dependencies ::= **dependences:**{*dependency*}
dependency ::= **strong_precedence_data** *ref_port* -> *ref_port;*
| **weak_precedence_data** *ref_port* -> *ref_port;*
| **precedence** *ref* ->*ref;*
| **data** *ref_port* -> *ref_port;*
ref_port ::= *NAME* | *NAME* **.** *NAME*
NAME ::= {'a'-'z' 'A'-'Z' '_'}+ {'a'-'z' 'A'-'Z' '_' '-' '0'-'9' '*'}

This is a simple example which is the corresponding SynDEx algorithm definition of the default operator in Signal language:

```
def algorithm default :
? int i1;
? bool i2;
? int i3;
! int o;
conditions : i2 =1;
references:
dependences:
   strong_precedence_data i1-> o;
conditions : i2 = 0;
references:
dependences:
   strong_precedence_data i3 -> o;
description:"x:=(i1 when i2) default(i3 when (not i2))"
```

Example 1: the definition of "default"

The SynDEx algorithm model is composed of several *algorithm* (or *sensor, actuator, constant, memory*) definitions. A *sensor* receives input signals from the environment and an *actuator* produces output signals to the environment; *constant* and *memory* definitions are special kinds of algorithm definitions which have more simple syntax. The main algorithm must be set according to the grammar "**main algorithm** *ref_type [arg_values];*".

From the sections above, we can see that SIGNAL and SYNDEX are both powerful environments used for real-time systems. Each of them used isolatedly has its specific features. The translation from SIGNAL to the SYNDEX formalism will give SIGNAL users access to all SYNDEX functionalities.

4 *SiSy* Translation Strategy

SiSy translation starts off from the internal abstract representation of a SIGNAL program to the input format of SYNDEX. It is not done directly from the SIGNAL code that we introduced in section 2, but from an arborescent hierarchical representation called *Hierarchical Conditional Dependency Graph* (HCDG) which is produced by the SIGNAL compiler.

4.1 The SIGNAL Hierarchical Conditional Dependency Graph

SIGNAL programs can be compiled into an HCDG which is a generalization of the Directed Acyclic Graph (DAG) and can be described with a six-tuple:

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

- $G =< N, \Gamma, I, O >$ is a dependency graph $< N, \Gamma >$ 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 representing clocks.
- $f_N : N \rightarrow C$ is a mapping each node to a clock; it specifies the presence condition of the nodes.
- $f_\Gamma : \Gamma \rightarrow C$ is a mapping each edge to a clock; it specifies precedence relations between the nodes.
- $\delta_\Gamma : \Gamma \rightarrow C$ is a mapping each edge to a clock while it specifies data dependencies between the nodes.

If $f_N(n) =$ `^c`, there exists a node $c \in N_c$ such that $f_N(c) =$`^c` and $f_\Gamma(c, n) =$ `^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 representing clocks. The boolean signal $c \in C$ represents the clock of a given signal x 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. $< C, \Sigma >$ defines a clock system organized as a *clock hierarchy*. It is a tree-like hierarchy defined by a function s: $s : C \rightarrow C \cup \{tick\}$, such that for every x in C, there is a single clock r, and a minimal integer n such that $s(s^n(x)) = s^n(x) = r$, *tick* represents the clock that is faster than the fastest clock of all signals in the compiled unit; r is called a root. A clock hierarchy tree can be seen as a representation of an inclusion relation between clocks. 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.

For a given program, different levels of representation of its clock hierarchy are distinguished. In fact, these different levels correspond to different phases of the compilation of a SIGNAL program. There are functionalities to change between some levels (e.g. from DC+ to bDC+ and from bDC+ to STS):

- In the DC+ level, clocks are represented as signals of the type *event*.
- In the bDC+ (boolean DC+) level, clocks are represented as boolean signals (no *event* type is used). Note that boolean signals representing clocks themselves have clocks represented as boolean signals; the clock hierarchy still exists in bDC+. It is provided with a single root, which is *tick*.
- The STS level (Symbolic Transition System), which is also called "flat bDC+", is a bDC+ level with a flat clock hierarchy. It has only one root, which is *tick*. State variables are defined at *tick*. Moreover, in the STS level, for every boolean clock signal b, $s(b)$=*tick*. This means there are at most two levels of clock hierarchy in the STS representation of a SIGNAL program.

For generating SYNDEX code, STS is the most appropriate level because in STS, all the clocks, which are the variables representing the control of the application, have been expanded up to 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*. This corresponds to the SYNDEX representation, which considers "remanent" variables.

4.2 Translation Rules

In a STS HCDG representation of a SIGNAL program, the clocks of the signals have been expanded so that the clock hierarchy tree contains at most two levels: the most frequent clock, the root of the tree (*tick* in *SiSy* also called *master clock*), and the children level, which comprises less frequent clocks defined by the *true* value of the boolean signals. Let us denote these booleans by $b_1, b_2, \ldots, b_n$, and the clock *when* b_i denotes the instants at which b_i is true. These clocks are children of the clock *tick*, they are also called *sub clock*s. Since during the SIGNAL compilation, each clock is associated with the sub-graph containing the nodes that are defined at this clock, the translation strategy is the following: whenever a graph is associated with a clock, this graph is translated as one *algorithm* in SYNDEX.

Translation of Clock Hierarchy. So any graph g_i associated with a clock that is a child of *tick*, will be translated to an algorithm following the *standard SiSy* translation scheme. In this algorithm, however, a condition is defined, corresponding to the values of the boolean representation of this *sub clock* b_i, which is an input of g_i, $b_i = 1$. The *standard SiSy* translation scheme for each graph will be presented in detail in the following subsection. The translation of g_i can be sketched as follows:

```
def algorithm P_b_i:
? ...
! ...
conditions: b_i = 1
SiSy(g_i)
```

The top level graph P contains the nodes associated with the clock *tick*, as well as process calls corresponding to each of the graphs associated with the *sub clock*s of *tick*. The *SiSy* translation of P is an algorithm P obtained in the *standard* way (in particular, it contains the translations of all the nodes of P, and those representing these process calls). The definitions of this algorithm are not conditioned (in SynDEx, *conditions: true*). The input and output signals (in SynDEx, they are called *ports*) are translated from the communication signals between clock graphs. Interface signals that communicate externally are *read /write* nodes in STS and get translated into sensors and actuators in SynDEx.

At last, after translating the *master clock* and all the *sub clock*s, the algorithm of the *master clock* is set to the main algorithm of the SynDEx program following the form "**main algorithm** *ref_type [arg_values];*".

Translation of Nodes. Most of *SiSy* effort is spent translating nodes. The way to translate nodes is just what we called "the *standard SiSy* translation" before. From the grammar and the example of SynDEx algorithm definitions (section 3.2), we can see that generally there are four parts in every algorithm: *ports, conditions, references*, and *dependences*. *Ports* and *conditions* are produced while translating the clock hierarchy. *References* and *dependences* are the main body of an algorithm definition. They are produced according to the attached nodes of the considered clock. In STS, the nodes can be classified into:

- **Constants:** The constants are explicitly represented by references to *constant* vertices in SynDEx (associated dependences follow trivially).
- **Equations:** For the equations, we consider a general definition `X := F(Y1, Y2, ..., Yn)` where F is some Signal n-ary operator. It is obvious that it is sufficient to only consider elementary expressions (using only one operator and adding auxiliary variables), because it is always possible to rewrite a definition containing composed expressions into a composition of elementary definitions. Such elementary operators include all the basic static *monochronous* operators (e.g., `add` or `mod`), the dynamic *monochronous* operators (e.g. `delay` or `window`), and some *polychronous* operators (e.g. `default` or `cell`). For such elementary operators, including state variables, SynDEx defines libraries to provide basic algorithm declarations (example 1 in section 3.2, is the declaration of the Signal "default" operator in the int library of SynDEx). We assume that `X := F(Y1, Y2, ..., Yn)` is an operation of type T, then there is an algorithm named F with type T declared in the predefined library. The *SiSy* translation of this equation is a reference to F associated with T, noted: `T/F`. This reference must have a unique name

in the algorithm in which it is contained, for example, this name can be the name of the algorithm, F, suffixed with the current value of a reference counter: `T/F F_k`. To specify the dependences, this occurrence is designated by its name `F_k`. There is another polychronous operator in SIGNAL: `when`. Considering the general form of a *when* equation, node N, `X := E when c`, `c` represents the clock of the considered definition of `X`. Since the boolean variable `c` has been considered when the node N was found in a subclock, it is always correct to translate the node N just like an ordinary node, `X := E`.

- **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*.
- **Partial definitions:** For the *SiSy* translation of partial definitions, we consider:
  `X ::= E`$_1$ `when c`$_1$
  ...
  `X ::= E`$_n$ `when c`$_n$
  The `c`$_i$ clocks are exclusive clocks, and the clock `c`$_j$ is the clock at which `X` is defined by the synchronous expression `E`$_j$. When translating such partial definitions, the original graph is rewritten as follows: At first, the nodes representing the list of partial definitions associated with the signal `X` are deleted from the attached clocks. Then, it is always possible to generate a new subclock which includes the signal `X` produced by all the nodes representing its partial definitions:
  `X := (E`$_1$ `when c`$_1$`) default ... default (E`$_n$ `when c`$_n$`)`
  The connections or dependencies involving `X` are consequently transferred to this node.
- **Process call:** The *SiSy* translation of a process (or sub-process) Q comprises exactly all the objects mentioned in this section. The process is defined explicitly as an *algorithm* named Q, then, a call to the process Q is translated into a reference to the algorithm Q, with the unique name `Q_k`.
- **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.
- **Subgraph:** the nodes and dependencies of a subgraph are subsets of the nodes and dependencies of the graph in which they are contained. So, they are handled in the *standard* way by the *SiSy* translation. If, for some reason, a subgraph has to be kept as a proper object, it has first to be transformed into a graph and replaced in its including graph by a process call to this new graph. Then, the *SiSy* translation is the same as for any process call.

For non elementary types that may be eventually used in a SIGNAL program, it is not possible to have the declarations before the translation; the corresponding library including the predefined operators in SIGNAL is produced by the translation program. Examples of such types are enumerated types, external types, etc. Arrays, which are also non elementary types, are treated separate-

ly since SynDEx provides an array constructor (one dimensional arrays). This constructor is used to represent the array types of the program. However, all array operators must be explicitly declared as algorithms.

Dependencies. After declaring the references for each node, dependencies must be set. The dependencies (the edges of the SynDEx graph) are built from:

1. The data flow connections of the Signal graph,
2. The explicit precedence relations.

With each data flow connection d: From the m^{th} output of a node p_k to the n^{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 m^{th} output of p_k, to the input of the vertex `SiSy`(p_l) corresponding to the n^{th} input of p_l. In SynDEx, a reference to some vertex `SiSy`(p_r) in a given *algorithm* must have a unique name: let us call `p_k`$_s$ and `p_l`$_t$ the unique names of the considered references to the vertices `SiSy`(p_k) and `SiSy`(p_l). If `i_n` designates the n^{th} input of the vertex `SiSy`(p_l), and `o_m` designates the m^{th} output of the vertex `SiSy`(p_k), the *strong precedence data* associated with d is:
`strong_precedence_data p_k`$_s$`.o_m -> p_l`$_t$`.i_n;`

An interconnection communication is a special case of data flow connection, and get translated into explicit inputs and outputs in SynDEx. Let us take inputs for example: If `i_n` denotes the n^{th} input of the vertex `SiSy`(p_l), and `i_m` denotes the m^{th} input of the graph, the *strong precedence data* associated with d is:
`strong_precedence_data i_m -> p_l`$_t$`.i_n;`

When the input of the graph is a `read` node, it is translated into a `sensor` of the same type. (It is similar for `write` with `actuator`.) Then the *strong precedence data* is from the reference of this sensor to the input of the vertex who uses the input signal. Thus, If `i_n` denotes the n^{th} input of the vertex `SiSy`(p_l) and `o` designates the output of the m^{th} vertex `sensor`, let us assume it has the name `s_m`, the *strong precedence data* associated with d is:
`strong_precedence_data s_m.o -> p_l`$_t$`.i_n;`

Another kind of dependency is used to express *precedence* relations. With each precedence relation in the graph, the *SiSy* translation associates a *precedence* defined as follows:

When an expression of explicit dependency in Signal appears in STS, that is, there is a node like:
`E`$_i$ `-->` `E`$_j$
this expression is translated directly into a SynDEx *precedence*.

In this case, if `E`$_i$ and `E`$_j$ are simple expressions, we use `SiSy(E)` to represent translation of the expression E in the *standard* way and the unique reference names are produced. The corresponding dependency in SynDEx is:
`precedence SiSy(E`$_i$`) -> SiSy(E`$_j$`);`
If `E`$_i$ and `E`$_j$ are expressions with labels in the syntax of Signal, for example:
`S`$_i$ `::` `E`$_i$

S_j :: E_j
S_i --> S_j
the corresponding translation in SynDEx is still the *precedence* of the results of translating the equations:
precedence SiSy(E_i) -> SiSy(E_j);

4.3 An Example

Let us see a short, but complete example: a watchdog. Its task is to monitor some resources. After one action (ORDER) is received, if there is not any other action occurring for longer than the preset time (COUNTER), ALARM is emitted. Moreover, the value of ALARM is the number of the time unit since the beginning of execution. This watchdog monitor promotes the efficient usage of the resources. Here, because of space limitations, we set clock constraints in order that all signals are synchronous. The Signal process is as follows:

```
process WATCHDOG =
{integer COUNTER;}
(? boolean ORDER;
 ! integer ALARM;)

(| CNT ^= ORDER ^= HOUR ^= ALARM
 | HOUR := (HOUR $ init 0) +1
 | ZCNT := CNT $ init 1
 | CNT := COUNTER when ORDER
       default ZCNT-1 when ZCNT >0
       default -1
 | ALARM := HOUR when CNT =0 default 0
 |)
where
   integer HOUR, ZCNT, CNT;
end
```

The corresponding SynDEx algorithm is :

```
syndex_version : "6.5.2"

include "Signal_int.sdx";
include "Signal_bool.sdx";

#-------------------------
def algorithm CLK_CNT_41_0 :
? bool  ORDER;
? int  CNT_37;
? bool  C_CNT;
? bool  C_70;
! int  CNT_41;
conditions: C_CNT = 1;
references:
  Signal_int/constante<5>  constante_3;
  Signal_int/default default_3;
dependences:
  strong_precedence_data default_3.o -> CNT_41;
  strong_precedence_data constante_3.o -> default_3.i1;
  strong_precedence_data ORDER -> default_3.i2;
  strong_precedence_data CNT_37 -> default_3.i3;

#-------------------------
def algorithm CLK_ORDER :
conditions: true;
references:
  Signal_bool/input input_1;
  Signal_int/default default_1;
  Signal_int/constante<1>  constante_1;
  Signal_int/n_delay<1>  n_delay_1;
  Signal_int/usub usub_1;
  Signal_int/default default_2;
```

```
  Signal_int/constante<0>  constante_2;
  Signal_int/n_delay<1>  n_delay_2;
  Signal_int/add add_1;
  Signal_int/sub sub_1;
  Signal_int/strictly_greate strictly_greate_1;
  Signal_int/equal equal_1;
  Signal_bool/logic_or logic_or_1;
  Signal_bool/logic_not logic_not_1;
  Signal_bool/logic_and logic_and_1;
  Signal_int/output output_1;
  CLK_CNT_41_0 CLK_CNT_41_0_1;
dependences:
  strong_precedence_data default_1.o -> output_1.i;
  strong_precedence_data add_1.o -> default_1.i1;
  strong_precedence_data equal_1.o -> default_1.i2;
  strong_precedence_data constante_2.o -> default_1.i3;
  strong_precedence_data constante_1.o -> n_delay_1.i1;
  strong_precedence_data default_2.o -> n_delay_1.i2;
  strong_precedence_data constante_1.o -> usub_1.i1;
  strong_precedence_data CLK_CNT_41_0_1.CNT_41 -> default_2.i1;
  strong_precedence_data logic_or_1.o -> default_2.i2;
  strong_precedence_data usub_1.o -> default_2.i3;
  strong_precedence_data constante_2.o -> n_delay_2.i1;
  strong_precedence_data add_1.o -> n_delay_2.i2;
  strong_precedence_data n_delay_2.o -> add_1.i1;
  strong_precedence_data constante_1.o -> add_1.i2;
  strong_precedence_data n_delay_1.o -> sub_1.i1;
  strong_precedence_data constante_1.o -> sub_1.i2;
  strong_precedence_data n_delay_1.o -> strictly_greate_1.i1;
  strong_precedence_data constante_2.o -> strictly_greate_1.i2;
  strong_precedence_data default_2.o -> equal_1.i1;
  strong_precedence_data constante_2.o -> equal_1.i2;
  strong_precedence_data input_1.o -> logic_or_1.i1;
  strong_precedence_data strictly_greate_1.o -> logic_or_1.i2;
  strong_precedence_data input_1.o -> logic_not_1.i1;
  strong_precedence_data logic_not_1.o -> logic_and_1.i1;
  strong_precedence_data strictly_greate_1.o -> logic_and_1.i2;
  strong_precedence_data input_1.o -> CLK_CNT_41_0_1.ORDER;
  strong_precedence_data sub_1.o -> CLK_CNT_41_0_1.CNT_37;
  strong_precedence_data logic_or_1.o -> CLK_CNT_41_0_1.C_CNT;
  strong_precedence_data logic_and_1.o -> CLK_CNT_41_0_1.C_70;

# main algorithm
main algorithm  CLK_ORDER;
```

It is easy to validate the translation by simultating the application in both environments. As expected, when executing the SIGNAL code and the SYNDEX code with the same input data, we get the same results.

5 Conclusion

Real-time systems are crucial components of controllers for planes, robots, cars, even basic household devices. Malfunction may be dangerous, and repairing is usually very costly. Therefore high competence of developers and powerful tools are needed for developing this kind of systems. SIGNAL and SYNDEX are both elegant and well defined integrated environments that can be used for the development of synchronous systems. But the two environments are isolated of each other, each having its own features. The *SiSy* translator offers an interface to reach SYNDEX functionalities from SIGNAL code. The main add-ons of this translator are the following:

1. It gives SIGNAL designs access to the functionalities of SYNDEX, in particular the possibility to get and to prototype distributed implementations obtained from quantitative criteria;
2. It enables SIGNAL as a possible input formalism for SYNDEX users;
3. It allows the use of SYNDEX on applications developed in formalisms for which it is of interest to have a SIGNAL intermediate representation.

In the POLYCHRONY environment, SYNDEX could be made in particular for the general method we propose for distributed code implementation[GL99]. In this method, an application is first described as a polychronous SIGNAL process, comprosed of atomic sub-processes, and target architecture is first viewed as a set of abstract nodes (tasks and processes) to which functional sub-processes are allocated. Currently, the allocation is manual, but it could be done with help of SYNDEX optimigation algorithms. The *SiSy* translator is still in development, but the prototype has been made available with the newest version of POLYCHRONY [E03]. Currently, we are concentrated on composite types translation, as well as testing and experimenting on realistic examples provided by the ACOTRIS project.

References

[A03] ACOTRIS Web site : *http://www.acotris.c-s.fr/*

[ABL95] P. Amagbégnon, L. Besnard and P. Le Guernic, Implementation of the Data-flow Synchronous Language SIGNAL, *PLDI'95 (Programming Languages Design and Implementation)*, 163-173, 1995.

[BD91] F. Boussinot, R. De Simone. The ESTEREL language. *Proceedings IEEE*, 79-9, 1991.

[BGL02] L. Besnard, Th. Gautier, P. Le Guernic, SIGNAL V4 -INRIA version: Reference Manual, IRISA, December 2002.

[BLSS87] A. Beneveniste, P. Le Guernic, Y. Sorel, M. Sorine, A denotational theory of synchronous communicating systems, *INRIA Research Report 685*, Rennes, France, 1987, Also appear in *Information and Computation.*

[E03] ESPRESSO Web site: *http://www.irisa.fr/espresso/Polychrony/*

[GL99] Th. Gautier, P. Le Guernic, Code generation in the SACRES project. Towards System Safety, Proceeding of the Safety-critical Systems Symposium, SSS'99, Springer, 1999

[H91] N. Halbwachs et al. The synchronous data flow programming language LUSTRE. *Proceedings IEEE*, 79-9, 1991.

[KS98] R. Koclik, Y. Sorel, A Methodology to Design and prototype Optimized Embedded Robotic Systems, *2nd IMACS International Multiconference CESA'98,* Hammamet, Tunisia, April 1998.

[M02] C. Macabiau, [illegible] 0 Tt9463 0 Td Tj 9.58926 0 Td (9.6 Td ([KS9876 0 Td (e-rTd (uni49j 12.478