A synchronous model of the IEC 61131 PLC languages FBD and ST in SIGNAL

Fernando Jiménez-Fraustro[1] and Éric Rutten[2]

[1] EP-ATR Group, IRISA, F-35042, Rennes, France
Fernando.Jimenez@irisa.fr, www.irisa.fr/ep-atr
[2] BIP Group, INRIA Rhône-Alpes, F-38330 MONTBONNOT, France
Eric.Rutten@inrialpes.fr, www.inrialpes.fr/bip

Abstract. Industrial control systems feature real-time programs embedded in a variety of machines and processes, showing complexity and safety criticality. Standardization has led to the definition of the IEC 61131 design standard for Programmable Logic Controllers (PLCs). Its formalization for purposes of analysis and tool support is an active research topic. The synchronous approach to real-time and reactive systems has resulted in formalisms and effective tools for the compilation, analysis and verification of specifications. With the motivation to give access to these formal techniques and tools, this paper presents a synchronous model of the PLC programming languages ST and FBD, and of the Program Organization Units (POU), based on the language SIGNAL.

Keywords: programmable logic controllers, IEC 61131 standard, formal model, synchronous languages, SIGNAL, tool support.

1 Motivations

The safe design of industrial controllers. Industrial control systems, e.g. in factory automation, feature real-time programs embedded in a variety of machines and processes. Their complexity is related to their connection with an environment which has its own dynamics, and is itself complex (involving control theory and the physics of the controlled process). This can show through the number of sensors to be taken into account, or of actuators to be controlled. The architectures upon which the controllers execute (meaning the hardware as well as operating system and execution platform) can be massively parallel, heterogeneous and connected through particular communication networks (e.g., CAN or field bus). The very diversity of these levels and aspects that interact sometimes deeply, and the interconnection between complex controllers, can increase the difficulty of their correct design.

Furthermore, they often control dangerous or safety critical activities. They can concern the security of persons (in the case of transportation systems, typically), the integrity of the environment (in energetic installation, e.g., nuclear) or material goods (e.g., in the reliability of production lines). It is therefore very important to assist as much as possible the validation of such applications. Certification authorities begin to advocate the use of formal and automated methods. Hence the importance of work on formal modelling for the design of industrial controllers.

These control systems are often implemented on Programmable Logic Controllers (PLC). Their design and implementation relies on industrial standards, like the ones emitted by the International Electrotechnical Commission, e.g., the IEC 848 norm on Grafcet [15], or the IEC 1131 norm[1], and more specially its IEC 1131-3 part concerning programming languages [17]. These various languages can be used to program in a modular way. Some of them are : ladder diagrams (LD), imperative sequential languages like Structured Text (ST), Sequential Function Charts (SFC). This variety reflects the levels of design and of cultures involved in the design of large controllers, where legacy specifications can be very important. The integration of these informal notations is treated in the norm at a syntactical level. A design assistance has to be provided to designers, giving clear semantics to large specifications, integrating the different aspects of a system that are expressed through different paradigms. Especially, the resulting behavior must be clear when several languages are used, according to their interoperation, and the relations between modules. Independently of this, even the volume of specifications can require a formal basis upon which an automated analysis can be achieved.

Therefore, an environment of practical tools with a formal basis is an essential assistance to a correct design and implementation.

The synchronous technology for reactive systems. In the area of reactive real-time systems, the emergence of the synchronous approach to real-time systems [11] was precisely proposing to focus on formal methods for which effective tools can be built, providing for practical, formally based design environments [12]. It features programming languages covering different paradigms (including ESTEREL [6] , LUSTRE [13] , SIGNAL [21] , ARGOS [26]). They are defined in terms of a mathematical semantics. It is the base of all the transformations and analyses that are used in the compilation or verification, from a specification until its implementation. The synchrony refers to a particular formal model, with a notion of logical instant, where composability of sub-systems is made simpler, and more efficient w.r.t. automated analysis tools. It can be seen as the abstraction of a *scan* of a cyclic system. Other reactive languages that can be related to synchronous languages are ELECTRE [30], and STATECHARTS as well as the other languages of STATEMATE [14]. Their programming environments, academic and commercial, feature diverse analysis and code generation tools, interacting through exchange formats like DC+ [31].

Some of these functionalities are:

- the verification of the coherence of the specification, with regard to its invariant or static properties (at all instants of the execution) or dynamical properties (concerning sequences of reactions, events or actions);

[1] Since then, their designation changed to IEC 60848 and IEC 61131.

- simulation, or off-line performance evaluation;
- the automated generation of executable code, in different target languages (e.g., C, Ada, or even more specific languages: circuit description languages, or PLC programming languages); this can take into account the possibly distributed execution architecture.

Hence, giving a synchronous model to a programming language, and giving it the concrete form of a translator to these formats, constitutes a means to offer access to these functionalities from specifications written in that language. One advantage is the applicability to specifications inherited from the past. Another one is the avoidance of having application designers having to learn another new language.

The industrialization of these languages and tools lead to the marketing of programming environments. Users are found in the aeronautics or power plant control industries. Technology transfer activities occurs in cooperations like e.g., the ESPRIT project SACRES [3, 10]. One of the activities in this project was the integration of the languages of STATEMATE with the DC+-based technology, through a modeling in SIGNAL [4]; this work influenced the work presented here.

The proposed approach to a design environment for PLCs. As we can see, adequate models and tools are available, but adaptation to the specificities of industrial control has yet to be done; as described earlier, this involves taking into account design languages in the PLC programming world, different from those considered in the framework of the SACRES project for example.

The goal of our work is to connect the design of PLC-based controllers with the synchronous formal model, technology and tools. This way, using the formalisms of the standard as specification support, an assistance can be offered to face complexity and security of the systems under design.

The method we follow in the work presented here consists in the study of the semantics of the specification languages for such controllers, and the construction of a synchronous model of their behavior. The integration between languages is done through translation of the languages into a synchronous format. Here, we focus on:

- the industrial automation design standard IEC 61131 [16], and the programming languages defined there [17], especially ST, FBD and POU.
- the synchronized data-flow language SIGNAL [21], taking advantage of its specificities, and of the exchange format DC+ [31] which is close to SIGNAL.
- the functionalities of code generation and simulation.

It can be noted that it is rather more usual to go from a data-flow language towards an imperative language, for implementation purposes. Here we go the other way around, from ST to SIGNAL, in order to have a good representation for analysis and transformation.

Related work. Languages, compilers, and tools for the design of PLC controllers for embedded systems exist in commercial distribution, but offer essentially simulation and direct code generation. They lack formal models upon which more substantial assistance can be built, with a clear integration of the different notations and of their semantics. A number of approaches exist to the formalization of PLC languages [24, 20, 19, 8]. They usually concentrate on one particular language of the norm, rather than on their interaction, i.e., on the formalization of its behavior rather than on the integration of their individual semantics. Some of the models include the study of temporal properties, induced by the presence of timers, and the duration of the scan or reaction step. Examples are the temporal-automata model of the language IL (Instruction List, which is close to assembly language) [25], or of LD (Ladder Diagrams) [29], and the static analysis techniques applied to IL [7].

The relation between the basic behavior of PLCs and the synchronous models has lead to work on the synchronous modeling of SFC or GRAFCET using SIGNAL or timed automata [23], with a formalization of both interpretations : with or without stability search. The search for stability defines the reactive behavior by taking into account all internal effects of each input, even the indirect ones, and only then making the next reaction. The way this is done in GRAFCET is quite close to the notion of *superstep* in STATEMATE [14], i.e., it involves an iteration of internal single-step reactions, until no further state change is observed. This can be modeled in the framework of a synchronous language, e.g. ESTEREL [32], or using the hierarchy of clocks in SIGNAL [4]. The synchronous model was defined in SIGNAL, and integrated in the tools environment through an automatic translation, giving access to verification, code generation and simulation tools. Also to be noted is the model of GRAFCET using the asynchronous reactive language ELECTRE [30].

The formalizations of PLC languages sometimes involve their re-definition in order to make a better use of modern computer science techniques. This is a rigorous approach in that it allows for the identification and resolution of ambiguities in the norm. But it has the drawback of making acceptance difficult because of divergence from standards actually in use. For example, S-GRAFCET [2, 9] is a proposal close to the SFC, but where the search for a stability is redefined in a strictly synchronous way. In the synchronous approach, it is in the semantics of composition, which is used at compilation time, that reactions are defined globally, with all effects of an input. Another example of "re-design" of PLC language concerns SFC [1], which is given a traces semantics and defined with "the goal of making the presence or absence of certain properties evident".

As a contribution to the treatment of the complete norm (i.e. having a formal model of all the languages and their inter-operation), we focus here on the languages Structured Text (ST) and Function Block Diagrams (FBD), and on Program Organization Unit (POU).

In the following, we take some space to make presentations of the essentials of PLC programming (in Section 2) and of the synchronous language SIGNAL (Section 3), before describing the model in Section 4, outlining the handling of FBD, and detailing ST, and finally conclude in Section 5.

2 Programming PLC controllers and the IEC 61131 norm

2.1 Normalization

In the field of PLC programming, efforts of standardization tend to promote the unification of all the involved languages and architectures, as well as the communications between them. Some are more concerned with specification (IEC (60)848: Grafcet), or distributed implementation (IEC (6)1499), or batch processes (IEC (61)512) ... In particular, the IEC (6)1131 standard concerns the programming of PLCs. It is defined in five parts: introduction, hardware, programming languages, manual and user's guide, and communications. The third part, IEC (6)1131-3 [17], is the one particularly of interest to us: it describes the programming languages and their semantics, informally. It is divided in two parts : the common elements, and the four different programming languages.

2.2 The common elements of the standard

The common elements concern everything that can be used in all the languages. Elementary data types and variables, with default initial values, can be used to build derived ones. Program Organization Units (POU) are the basic structuring units. At a specification level called configuration, they can be associated to tasks of the execution platform. A graphical language called Sequential Function Charts is meant for the specification of sequencings of actions upon transitions depending on the occurrence of events and conditions.

A POU can be either:

- a function, which has no remanent internal state;
- a Function Block (FB), which does have internal data variables, hence a remanent state;
- a program, which is a FB with input/output interface connected directly to the environment.

A POU can be invoked textually in a ST program or SFC action, in a data-flow context in FBD, or directly, by a task at configuration level.

Functions and programs are particular cases of Function Blocks, hence the PLC programs are mainly structured by means of function block units. Function blocks are instances of function block types that encapsulate local data and algorithmic behavior and respect definite communication interfaces through strongly typed input/output variables. Their basic semantics involves cyclic repetition of a *scan*, where inputs are evaluated, a reaction is computed and outputs are emitted.

2.3 The four programming languages.

The algorithmic behavior of a function block is defined in one of the four languages of the standard. There are two graphical languages: Ladder Diagrams (LD), that are based on the graphical presentation of Relay Ladder Logic, and

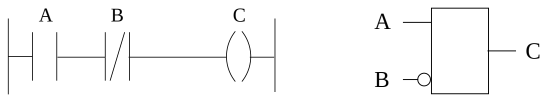
Function Block Diagrams (FBD), which express the behavior of functions, function blocks and programs as a set of interconnected blocks with the flow of variables between them. The two others are textual languages: Instruction List (IL), which resembles assembly languages, and Structured Text (ST), which is a sequential imperative language, of the family of Pascal, Ada, C, and the like. All four languages are interlinked through the programming units, this means that one can implement a function block in one language and use it in another. The syntax and semantics of all the languages allow the use of programming units described in any other language.

```
LD   A
ANDN B
ST   C
```

Instruction List (IL)

```
C := A AND NOT B
```

Structured Text (ST)



Ladder Diagrams   Function Block Diagram

Fig. 1. The four programming languages of the IEC 61131-3 norm.

3 The Signal language and programming environment

3.1 The Signal language.

Signal is a data-flow language, where programs are written in the form of systems of equations, composed in a block-diagrammatic fashion [21]; experiments illustrate the ways it can be practically used, and the actual meaning of the synchrony [22]. The basic objects are signals, which are series of typed values, with a clock, designating the set of instants at which they are present. The programs are equations relating the values of signals at a given instant. For a signal `X`, its clock is noted $\widehat{X}$; for a Boolean `B`, $[B]$ is the set of instants where it is present and true. We have set relations on clocks: $\cap$ (intersection), $\cup$ (union), $\subseteq$ (inclusion), $=$ (set equality) and $\setminus$ (set subtraction). The primitives of the language are used to specify relational constraints on the clocks of signals, and functional definition of values.

The primitive constructs are as follows:

- functions: e.g. on Booleans: `X := A and not B`, where $\widehat{A} = \widehat{B}$;
- selection (or down-sampling): `X := Y when B`, where $\widehat{X} = \widehat{Y} \cap [B]$;
- merge (or up-sampling; priority to `Y`): `X := Y default Z`, where $\widehat{X} = \widehat{Y} \cup \widehat{Z}$;

– and finally, delay (previous value, with initialization): `X := Y$1 init X0`, where $\widehat{Y} = \widehat{X}$.

They can be composed using the composition operator noted "`|`"; SIGNAL is a declarative equational language, hence the order in which equations are written has no influence on the result. A SIGNAL program is a system of equations on signals, and its behavior is defined as the set of solutions of the constraints. There exist derived constructs for comfort and structuring e.g., the synchronization constraint (`^=`), and external (e.g. C) functions. They can be used to express behaviors in a way reminiscent of sequential circuits, with delays used as registers.

A difference is that the down- and up-samplings allow for not strictly single-clocked specifications as in circuits, hence enabling less constrained clocks, more modularity and reusability.

```
process decounter = (? integer X;
                     ! integer Y )
(| Y := X default ZY - 1
 | ZY := Y$1 init 0
 | X ^= when (ZY=0)
 |);
```

X	3				5						7	...
Y	3	2	1	0	5	4	3	2	1	0	7	...
ZY	0	3	2	1	0	5	4	3	2	1	0	...

Fig. 2. A SIGNAL program with up-sampling : a process de-counting its inputs.

3.2 A specific feature: up-sampling.

As an illustration, we can take the decounter shown in Figure 2. Its internal activity is faster than its inputs. In other words, its internal clock includes that of inputs. In the first line, it defines `Y` by either accepting input `X` when it is present, or decrementing the value of `ZY`. The second line defines `ZY` as the preceding value of `Y`, kept through a delay (which is the only form of memory in SIGNAL). Going down all intermediate values leads to `0` on signal `Y`, as illustrated in Figure 2. In the third line, the synchronization constraint (`^=`) specifies that, at the instants when `ZY` is `0`, we take or accept a new input `X`.

Hence it is the internal state of the process that decides on synchronization with the environment. This mechanism will be used in the modeling of ST, in order to take into account unbounded loops, and their expansion in series of instants between inputs reception and outputs production.

3.3 The SIGNAL environment.

The SIGNAL environment integrates the tools implemented as a result of the studies around SIGNAL and its model, and is organized around the DC+ format and virtual machine. Figure 3 mentions its functionalities, documented in the referenced literature. It has a bloc-diagrammatic graphical interface. The compiler is a multi-faceted tool box for the analysis and gradual, parameterized transformation of the internal representation of programs (a hierarchical conditioned dependencies graph) [5]. It handles modular compilation, optimization, code generation for sequential or distributed execution platforms, hardware/software co-design. It is related to a formal tool based on model-checking techniques, performing verification and discrete controller synthesis [27]. One feature (not shown in the Figure 3, but used in an experiment [18]) is a connection to the Simulink tool for simulation of hybrid systems. A commercial version of the environment called SILDEX is made and distributed by TNI (`www.tni.fr`).

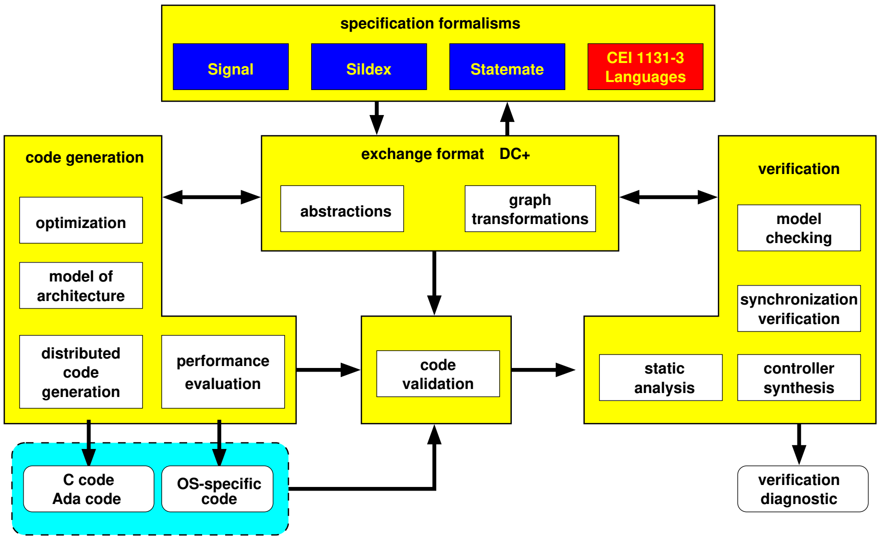


Fig. 3. The SIGNAL programming environment around DC+.

3.4 The DC+ open environment.

The exchange formats are a basis upon which the architecture of such programming environments can be re-thought in a more open way, where integration can be effectively achieved:

- of formalisms, through their formalization in terms of a common model: here, the synchronous models can be encoded in DC+;
- of tools, through their modularization in such a way that they can accept models in the concrete exchange format.

The DC+ open programming environment is an instantiation of a design environment based on that technology [10] (`www.tni.fr/sacres`). It also integrates a verification technology with a SIEMENS model-checking tool and STATEMATE [14] is integrated to the synchronous technology by providing a translator to Signal and Dc+ [4]. This examplifies that it is possible to give access to synchronous technology from languages originally not defined within the synchronous approach. This potentiality is extensible to languages which show reactivity, and is especially meaningful when safety of the design is crucial, and complexity requires automated assistance to its assessment. This is the case of industrial controllers programming, hence our approach. In particular, the language of actions in Statemate [14] features imperative sequential constructs quite close to those of ST.

Compared with previous work, the approach presented here continues the extension of the accessibility of synchronous techniques, first experimented with directly with the Signal language [22], then shared with the other synchronous languages through exchange formats like Dc+ [5], and later, as a result of the Sacres project [3, 10], integrating the languages of StateMate [4]. In the work presented here, we follow the same method to go towards an integration of a different set of languages, and we also come in contact with an application domain more manufacturing-oriented than those of Sacres. An application and demonstration of this concerned the integration of an ST program and Simulink simulation code through the Signal-Dc+ environment [18].

4 Modeling IEC 61131-3 languages in Signal

The issue here is to construct a model in the form of a transition system representing the behavior of the IEC 61131-3 programs. The synchronous models present adequate basic aspects (notions of events and reactivity). The *scan* and input-output mechanisms are not as primitive i.e., they are defined in more elaborate way; they can be encoded in a synchronous model. Using a synchronous language, compared to generating directly a transition system as done in related work (see references in Section 1, presents the advantages of using a structured language, with constructs for the hierarchical organization of the model. Furthermore, it makes it possible to use the formal computations performed by the compilation, notably the clock calculus building a hierarchy of conditions, and the optimizations in state space.

4.1 Outline of the model of FBs and FBD

Function Blocks A Function Block is modeled as a Signal process, featuring the input/output interfacing ports, with additional control signals: an activation clock (the instants at which a computation begins), a basic clock (up-sampled w.r.t. the activation), and an output event indicating the end of the computation.

When a function block is activated, input data is passed to its input variables and its algorithm is executed. All variables are accessible for the algorithm i.e., the input and the local data, but also the output. The latter can be assigned several times within a scan, but it can be read by the calling context only after the activation is finished. The local data can be used to record relevant information for future use. All variables are defined with an initial value, either declared explicitely, or given by default according to type. They all persist from one activation to the next. The relation with the scan is done through ports: at activation, values of input ports are transfered to input variables, and at the end of the computation, values of output variables are transfered to output ports. Hence, a change of value on an input port during a computation does not influence it, and intermediate values of output variables are not sensed through the output port.

The activation clock is given by the calling context (sequential control in ST, presence of inputs in FBD, timers elapsing, etc). Upon this event, values of input ports are transfered in a register encoding each variable, and the algorithm is run, which might take more than one synchronous instant (in which case the basic clock is used, see Section 4.2). At its completion, output values are delivered, and the end event is emitted.

We have to manage the fact that input to the FB can arrive at different instants, and that they have to be kept until the next activation. To this end, SIGNAL processes working as a register are built: the value of their input signal is kept, and can be read at the clock of the output signal. It can be noted that the SIGNAL compiler performs a synchronization analysis such that it can in some cases detect equality of clocks. In these cases, it automatically performs the optimization consisting in removing the memorization (i.e., the use of the delay operator $). Here, it means that if inputs ports of the FB are always read together, at the instant where they arrive, then they do not need to be kept in a register, because there is no access to the input value between two occurrences of the input signal.

So, we have an automated construction of a model which can also be automatically optimized. This is a particular example of how the model can be built purposefully to make a better use fo the SIGNAL design environment.

So, Function Blocks are modeled by a SIGNAL process with such an interface, with register memorization of inputs, and their reading at activation clock to transfer the values to the body, and with a memorization of variables defined at that level.

The body of the process is the model of the body of the FB, to which activation and basic clock are given, and which produces end event. The body is itself modeled as a SIGNAL process, according to structural translation rules for the different constructs of the language e.g., ST.

Function Block Diagrams For a FB Diagram, which is a data-flow network, a simple case to consider is the one where all FBs in the FBD are instantaneous, i.e. do not require several SIGNAL instants to perform their computation: then, SIGNAL synchronous composition takes care of interconnection of ports, data dependencies, and evaluation ordering.

If one or more elements of the diagram may take more than one SIGNAL instant, typically because of unbounded loops in their body (see Section 4.2), then the flows of data have to be synchronized. A *rendez-vous* between the inputs of a FB decides on its activation: values are taken into the input registers when they are produced, and the last one arriving determines activation. Also, the end of computation of the FBD is derived from the end of activity of each FB.

4.2 Modeling Structured Text in SIGNAL

General principles We consider ST programs encapsulated in a Function Block (FB), with local data that persist from one activation to the next, and the algorithm that is executed during an activation. So a function block consists of an environment together with an evaluation function. The first is the state of all variables accessible to the function block. The latter is a statements list in the ST language, working on that environment. Statements can be assignments, `RETURN` which exits the whole program (without executing the instructions possibly in sequence), sequence of statements, and conditional (`IF` *expr* `THEN` *inst* `END_IF`). There is also a choice of several kinds of loops in ST: bounded (`FOR` *expr* `DO` *inst* `END_FOR`) and unbounded (`WHILE` *expr* `DO` *inst* `END_WHILE` and `REPEAT` *inst* `UNTIL` *expr* `END_REPEAT`), with an `EXIT` statement causing the current level of nested loops to be exited.

In imperative languages like ST, every assignment statement changes the environment, and several assignments can change the same variable during the execution. The translation of most of imperative ST into equational (data-flow) SIGNAL is quite classical, in that it builds a graph representing basically the data dependencies between instructions. SIGNAL considers signals which carry only one value per logical instant. Hence, in order to model in such a synchronous, single-assignment formalism a language with multiple-assignment variables, one can define *one signal for each assignment* to a variable. This is possible especially when we have a bounded sequence of them, i.e. a ST program with no unbounded loop. Therefore, we manage an environment *Env* with a signal associated to every state variable, passing it along the sequence of instructions. It can be seen as unfolding the sequence or bounded loop.

One can define, alternately, *one instant for each assignment* to the variable: this approach can be named *temporal expansion*. In this case, one has to define the control automaton of the ST program, describing the sequence between slices with no more that one assignment for each variable referenced. Then, one has to associate equations for each of the slices, and activate them at the right logical instant, according to the control automaton. The memorization of variables has to be managed from one instant to the other, in an explicit manner (in SIGNAL, using the `$` delay operator). The presence of unbounded loops in ST makes the use of this technique necessary.

The mechanism of SIGNAL called upsampling consists in being able to write SIGNAL programs that perform actions at more instants than the occurrences of an input signal, i.e. at a relatively faster clock (in the sense of sets of instants, the upsampling includes the upsampled). It is possible to specify at which instants of the internal clock (according to the internal state) new inputs can be acquired. In that sense, SIGNAL is a *proactive* language rather that a reactive language. This corresponds to what we want to model in ST: unbounded loops can insert an arbitrary number of instants between input acquisition and output emission (i.e. instants *inside* a *scan*). The moment when the loop is terminated (because of the condition or of an `EXIT` statement) is determined by the internal state of variables and control. Our model on ST in SIGNAL makes use of these two approaches. Its modeling benefits from results of the modeling into SIGNAL of the imperative language of actions in STATEMATE [4, 28].

Bounded instructions The general form of the translation function is

$$tran(Env_{input}, Env_{output})(stat)$$

with the statement to translate *stat*, the input environment Env_{input} and the new one Env_{output}, output of the translation of *stat*. The resulting SIGNAL code is shown in a box.

Assignment. For a statement `X:=`*expr*, a new signal is created as well as a new environment with this signal in it (substituting the previous one). The value carried by the signal will be the result of the translation to SIGNAL of *expr*, e.g.:

$$tran(\{\texttt{x1},\texttt{y1}\},\{\texttt{x2},\texttt{y1}\})(\texttt{X:=X+1}) = \boxed{\texttt{x2:=}\boxed{tran(\{\texttt{x1},\texttt{y1}\},\{\texttt{x1},\texttt{y1}\})(\texttt{X+1})}}$$

which boils down to $\boxed{\texttt{x2:=x1+1}}$

Sequence. It is noted "`;`" in ST. The environment calculated for the first statement is passed as input environment to the following statement:

$$tran(Env_{input}, Env_{output})(stat;statl) = \boxed{\begin{array}{l} tran(Env_{input}, Env_{stat})(stat) \\ \mid tran(Env_{stat}, Env_{output})(statl) \end{array}}$$

Conditional statement. The *if-then* statement has an input environment and must produce an output environment which can be either the same input environment when the condition is *false* or a new one produced by the statement list when it is *true*:

$tran(E_{in}, E_{out})(\texttt{IF}\ exprC\ \texttt{THEN}\ statlist\ \texttt{END\_IF}) =$

$$\boxed{\begin{array}{l} \quad \texttt{cond :=}\ \boxed{tran(E_{in}, E_{in})(exprC)} \\ \mid\ Env_{bodyif}\ \texttt{:=}\ E_{in}\ \texttt{when cond} \\ \mid\ \boxed{tran(Env_{bodyif}, Env_{endbodyif})(statlist)} \\ \mid\ E_{out}\ \texttt{:=}\ Env_{Endbodyif}\ \texttt{default (}E_{in}\ \texttt{when not cond)} \end{array}}$$

The conditional expression *exprC* is evaluated using the input environment. If it is true, a new environment (Env_{bodyif}) is created holding the signals with the clock `when cond`. The statement is translated using this environment, yielding $Env_{endbodyif}$. Then, Env_{output} is the merging of this one and the input one when the condition is *false* (and no statement is executed).

Bounded loop. As mentioned above, bounded loops can be unfolded into a sequence, and then treated just like the sequence presented before. In case of bounded loops with high bounds (i.e. numerous iterations) this unfolding or expansion can indeed be costly; one can discuss whether this is reasonable according to the cases under study.

Unbounded instructions We have to introduce new elements, and redefine the translation function in order to take into account the intermediate values taken by signals during the iteration. A memory has to be explicitely managed to carry values from one synchronous instant to the next, which is necessary in the framework of temporal expansion. A module controlling the activity of instructions has to assign an execution clock to each of them. The synchronization of inputs from the environment with the state of the controller will rely on the use the up-sampling mechanism of SIGNAL.

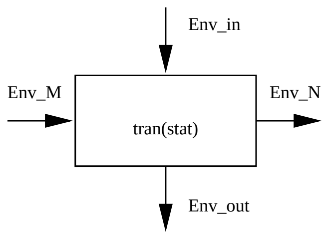


Fig. 4. General SIGNAL process structure produced by the translation function.

General translation function. In the general translation function, supplementary parameters giving access to the memory, as shown in Figure 4, are Env_M (memorized from the previous *scan*) and Env_N (computed anew in the current *scan*). As we will see, signals in Env_N are never present at the same time as the output environment Env_{out}.

Modeling memory management. Concerning memory management, all variables in ST have to be initialized. This is encoded as: $\boxed{x_M \texttt{ := } x_N\texttt{\$ 1 init } V_{init}}$ where $x_M \in Env_M$ and $x_N \in Env_N$ represent the same ST variable `X`. In the following, we note x_i its representing signal in Env_i.

Modeling control. It consists in determining the instants when iteration continues with computation on memorized values, and those when new input is accepted. Figure 5 shows the SIGNAL process with three inputs: Env_{in} the input environment, Env_M the memory environment, `end_body` the event designating the end of the iteration body. The output Env_{out} is forwarded to other instructions in sequence. The sub-module `CTRL_CYCLE` generates event `clk_M` one instant after receiving `end_body`. For this, a Boolean state variable is used: when it is true a new event `clk_M` has to be emitted, when it is false then it means that an inner iteration is busy or the iteration is ended. The sub-module `CTRL_ENV` takes as input `clk_M`, Env_{in} and Env_M. Its output Env_{out} will take values of Env_{in} on the first instant of the iteration, and of Env_M on the following instants when `clk_M` is present. This condition is for nested loops: a value for Env_{out} is produced only on the first instant of the body of the iteration.

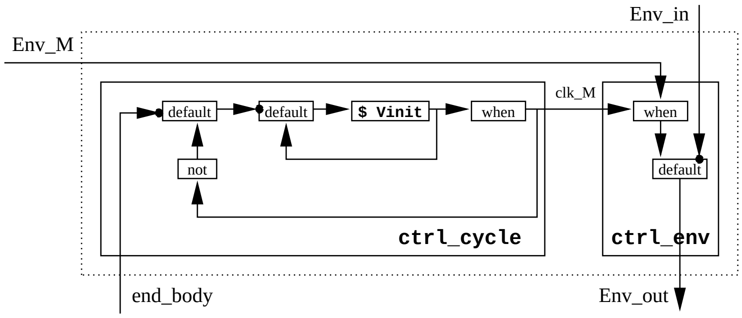


Fig. 5. Modeling cycle control

A simplified model of a function block (e.g., without management of interface variables) is illustrated in Figure 6.

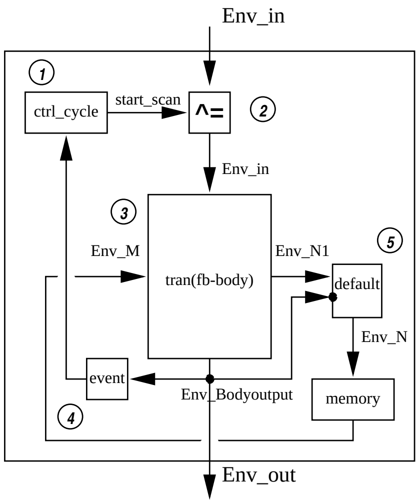


Fig. 6. Modeling a function block

Using up-sampling. It is done like in Section 3 and Figure 2, a synchronization constraint (`^=`) controls the acceptance of inputs from the external environment, one instant after the end of the body: $\boxed{x_{in}\ \texttt{\^{}=}\ clk}$ where clk is produced by the execution control system, as in point 2 of the following function block model.

Modeling a function block. A partial model of a function block is:

1. the state variable of CTRL_CYCLE is initially `false`, hence event `start_scan` is present at the first instant. In the following, it will be emitted one instant after body end (see point 4).
2. inputs are synchronized with `start_scan`, defining the initial environment.
3. with the latter, the translation of the body of the function block (i.e. the ST program) is activated.
4. when outputs of the block are present, event `end_scan` is generated, which will initiate a new *scan* on the next instant.
5. current values of signals to be memorized Env_N are obtained by merging outputs Env_{out} when present and intermediary values Env_{N1} computed at each instant by the body.

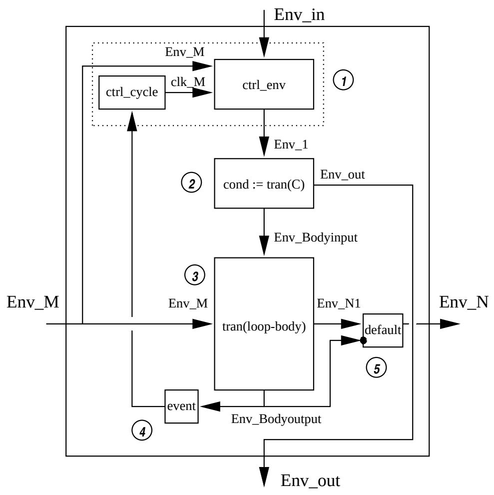


Fig. 7. Modeling a WHILE loop

Modeling a WHILE *loop.* It is illustrated in Figure 7:

1. the state variable of CTRL_CYCLE is here initially `false`, hence execution of the loop will be activated only on instants when input Env_{in} is present, giving its first value to Env_1. Afterwards, at the end of the loop body (see point 4), values are taken from the memory under control of event `clk_M`.
2. the condition is evaluated. When false, iteration ends; output environment Env_{out} is produced: $\boxed{x_{out} \texttt{ := } x_1 \texttt{ when not cond}}$ otherwise an environment is produced for the body: $\boxed{x_{Bodyinput} \texttt{ := } x_1 \texttt{ when cond}}$.
3. the translation of the body of the loop is executed.
4. at its end an event is given to CTRL to construct a new environment Env_1.

5. values currently computed Env_N are memorized (see point 1). They can come from either the end of the body ($Env_{Bodyoutput}$), or from a nested loop inside the body (Env_{N1}), i.e.: $\boxed{x_N := x_{Bodyoutput}\ \texttt{default}\ x_{N1}}$

Block exit statements The language ST features block exit statements: `EXIT` exits the current loop (bounded or unbounded), and `RETURN` exits the whole ST program, at the level of its encapsulation in a function block. They both involve:

- inhibiting the sequel of the sequence of instruction in the block,
- returning the right environment, i.e., the variables with the values at the point of exit.

This is managed by both local translations schemes and additional features in the function block body model.

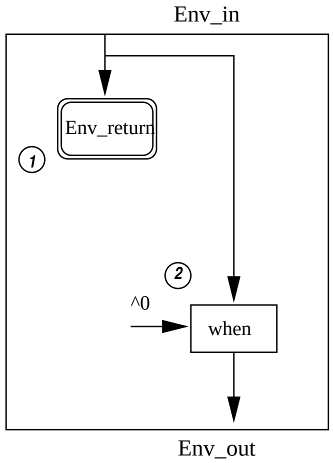


Fig. 8. Modeling a `RETURN` statement

Modeling a `RETURN` *statement.* It is illustrated in Figure 8, where both aspects are taken into account:

1. the environment Env_{in} is transmitted to a special environment: Env_{return} that will be managed by the function block body model, in order for that valuation of the variables to be produced as output of the program (see below, and Figure 9).
2. the output environment Env_{out} is produced with the null clock (^0). This way, all statements following the `RETURN` statement in the sequence are disabled.

The double line in the Figure means that Env_{return} is made available at the function block body level (without complexifying notations in the structural translation presented before).

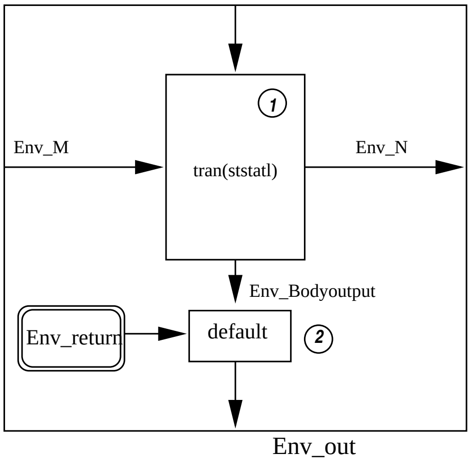


Fig. 9. Modeling a function block body

Modeling the function block body. It is illustrated in Figure 9:

1. the function block body is translated as a list of ST statements (*ststatl*).
2. the body can end either:
   - by the execution of the last statement of the list, in which case the output $Env_{Bodyoutput}$ is present;
   - or by the execution of a `RETURN` statement, in which case the environment Env_{return} is present.

   As the `RETURN` statement inhibits any other instruction in the sequel of the sequence, the respective presences of $Env_{Bodyoutput}$ and Env_{return} are exclusive. Given that ST is a sequential language, there can be only one `RETURN` statement encountered (the first one) at each execution of the body, hence only one environment Env_{return} produced. As in Figure 8, it is shown in a double line meaning that it is made available at the function block body level.

   It is merged with the $Env_{Bodyoutput}$ to produce the function block body output environment Env_{out}.

Modeling the `EXIT` *statement.* It is managed quite similarly to the `RETURN` statement, but with a different scope.

A special environment Env_{exit} is produced by each `EXIT` statement. The difference with above is where these environments are processed. For example, in the `WHILE` loop translation of Figure 7, a new output environment Env_{newout} must be produced by mixing the old output Env_{out} with Env_{exit}, meaning that the loop may be finished by two ways : normal termination (when the condition is false) or by the execution of an `EXIT statement`.

5 Conclusion and perspectives

We approach the integration of the languages Structured Text (ST) and Function Block Diagrams (FBD) of the IEC 61131 norm with the synchronous technology by modeling the languages into SIGNAL equations. The up-sampling of SIGNAL is used to model time internal to the PLCs execution cycle, illustrating the use of synchronous models to represent non-instantaneous input-output behavior. The use of the functionalities of the SIGNAL environment has been illustrated by an application in hybrid simulation, using the connection between SIGNAL and Simulink, of a continuous/discrete control system consisting of three pumps [18]. The model is implemented as an automatic translator. Its validation in a formal manner is made difficult by the absence of a formal semantics of the IEC 61131-3 norm in the beginning. However, the structural construction of the model has been carefully argumented. One technique would be to define a very simple model, encoding directly a transition system from the programs, and then to establish some equivalence with the more elaborate and optimized model that we propose. Having an automated generation of the simple model and equivalence checking at each translation would be an alternative to proving the model construction correct in general.

The interest of this work lies in the study of the relation between different formalisms in reactive languages, in the line of corresponding work concerning STATEMATE [4]. It is different in that it concerns another set of languages, with different features, and with a different application domain: manufacturing systems rather than embedded systems (automotive, aeronautical, etc). As such, it widens the application of synchronous techniques. Perspectives are in extending the modeling of the elements of the IEC 61131 norm (e.g. integrating complex data types, modeling other IEC 61131 languages, and taking care of inter-operability questions), validating it on more applications, making actual use of the synchronous analysis and compilation functionalities, adapted to the specificities of the IEC languages and PLC architectures, and generalizing the use of upsampling to refine a reaction instant into a sequential implementation.

References

1. S. Anderson and K. Tourlas. Design for proof: An approach to the design of domain-specific languages. *Formal Aspects of Computing*, 10:452–468, 1998.
2. Charles André and Daniel Gaffé. Proving properties of GRAFCET with synchronous tools. In *Proceedings of IEEE-SMC Computational Engineering in Systems Applications (CESA), Lille, France, July 1996*, pages 777–782, 1996.
3. Philippe Baufreton, Xavier Méhaut, and Eric Rutten. Embedded systems in avionics and the SACRES approach. In *Proceedings of The 16th International Conference on Computer Safety, Reliability and Security, SAFECOMP'97,* York, United Kingdom, September 7–10, 1997, pages 311–320. Springer, 1997.
4. Jean-René Beauvais, Thierry Gautier, Paul Le Guernic, Roland Houdebine, and Eric Rutten. A translation of STATECHARTS into SIGNAL. In *Proc. of the Int. Conf. on Application of Concurrency to System Design, CSD'98,* Aizu-Wakamatsu, Japan, March 23–26. IEEE, 1998.

5. A. Benveniste, T. Gautier, P. Le Guernic, and E. Rutten. Distributed code generation of dataflow synchronous programs: the sacres approach. In *Proceedings of the 11th Int. Symp. on Languages for Intensional Programming, ISLIP'98, Sun Microsystems, Menlo Park, California (USA)*, May 1998.
6. Gerard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.
7. Sébastien Bornot, Ralf Huuck, Yassine Lakhnech, and Ben Lukoschus. Utilizing static analysis for programmable logic controllers. In *Proceedings of the* 4th International Conference on Automation of Mixed Processes, ADPM'00*, 18-19 September 2000, Dortmund, Germany*, 2000.
8. A. Fett, G. Egger, and P. Pepper. Formal specification of a safe PLC language and its compiler. In *Proceedings of the 13th Int. Conf. on Computer Safety, Reliability and Security, SAFECOMP'94,* Anaheim, October 1994, 1994.
9. Daniel Gaffé. *Le modèle* GRAFCET *: réflexion et intégration dans une plate-forme multi-formalisme synchrone.* Thèse de Doctorat en Informatique, Université de Nice-Sophia-Antipolis, January 1996. (In French).
10. Alvery Grazebrook. Sacres - formalism for real projects. In F. Redmill and T. Anderson, editors, *Safer Systems.* Springer-Verlag, 1997.
11. Nicolas Halbwachs. *Synchronous programming of reactive systems.* Kluwer, 1993.
12. Nicolas Halbwachs. Synchronous programming of reactive systems – a tutorial and commented bibliography. In *Proceedings of the International Conference on Computer-Aided Verification, CAV'98,* Vancouver, Canada, june 1998, number 1427 in Lecture Notes in Computer science (LNCS). Springer-Verlag, 1998. LNCS nr. 1427.
13. Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE,* special issue on *Another Look at Real Time Programming*, 79(9):1305–1320, September 1991.
14. David Harel and Amnon Naamad. The STATEMATE semantics of STATECHARTS. *ACM Trans. on Software Engineering and Methodology*, 5(4):293–333, Oct. 1996.
15. IEC. Specification language GRAFCET for sequential function charts. Technical Report IEC 848, IEC (International Electrotechnical Commission), 1988.
16. IEC. International standard for programmable controllers. Technical Report IEC 1131 parts 1–5, IEC (International Electrotechnical Commission), 1993.
17. IEC. International standard for programmable controllers: Programming languages. Technical Report IEC 1131 part 3, IEC (International Electrotechnical Commission), 1993.
18. Fernando Jiménez-Fraustro and Eric Rutten. Hybrid simulation of IEC-61131 PLC programs using Signal and Simulink. In *Proceedings of the* 4th International Conference on Automation of Mixed Processes, ADPM'00*, 18-19 September 2000, Dortmund, Germany*, 2000.
19. B. Kramer and W. A. Halang. Achieving high integrity of process control software by graphical design and formal verification. *Software Engineering J.*, Jan. 1992.
20. S. Lamperiere-Couffin, O. Rossi, J.J. Lesage, and J.M. Roussel. Formal validation of PLC programs: a survey. In *Proceedings of European Control Conference 1999 (ECC'99),* Karlsruhe, Germany, 31. August - 3. September 1999, 1999.
21. Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE,* special issue on *Another Look at Real Time Programming*, 79(9):1321–1336, Sep. 1991.

22. Paul Le Guernic and Eric Rutten. Experiments with the synchronous methodology illustrating its support of predictability. In *Proceedings of the 21^{st} IFAC/IFIP Workshop on Real Time Programming, WRTP'96,* Gramado, RS, Brazil, November 4–6, 1996, pages 81–86. Elsevier, 1996.
23. Philippe Le Parc, Dominique L'Her, Jean-Luc Scharbarg, and Lionel Marcé. Grafcet revisited with a synchronous data-flow language. *IEEE Transactions on Systems, Man and Cybernetics - Part A: Systems and Human*, 29(3), May 1999.
24. A. Mader. A classification of plc models and applications. In *Proceedings of the Workshop on Discrete Event Systems, WODES 2000, Gent, Belgium, August 21-23*, 2000.
25. A. Mader and H. Wupper. Timed automaton models for simple programmable logic controllers. In *Proceedings of the 11th Euromicro Conf. on real-Time Systems, ECRTS'99,* June 9–11, 1999, York, England. IEEE Publ., 1999.
26. F. Maraninchi and N. Halbwachs. Compiling argos into boolean equations. In *Formal Techniques for Real-Time and Fault-Tolerance (FTRTFT)*, pages 72–90, Uppsala (Sweden), September 1996. Springer Verlag, LNCS nr. 1135.
27. H. Marchand and M. Samaan. On the incremental design of a power transformer station controller using controller synthesis methodology. In *Proceedings of the World Congress on Formal Methods, FM'99, Volume II, Toulouse, France, September 20-24, 1999*, number 1709 in Lecture Notes in Computer Science (LNCS), pages 1605–1624. Springer Verlag, 1999. LNCS nr. 1709.
28. Mirabelle Nebut. Modélisation de StateMate en Signal : le langage impératif des actions. DEA en informatique, IFSIC, Université de Rennes 1, Sep. 1998.
29. O. Rossi and Ph. Schnoebelen. Formal modeling of timed function blocks for the automatic verification of ladder diagram programs. In *Proceedings of the* 4th International Conference on Automation of Mixed Processes, ADPM'00*, 18-19 September 2000, Dortmund, Germany*, 2000.
30. O. Roux and V. Rusu. Translating from GRAFCET to the reactive language ELECTRE. *Automatique, Productique et Informatique Industrielle*, 28(2), 1994.
31. SACRES Consortium. The common format of synchronous languages – the declarative code dc+ version 1.4. Technical report, Esprit project EP20897 SACRES, 1997. (`www.tni.fr/sacres`).
32. S.A. Seshia, R.K. Shyamasundar, A.K. Bhattacharjee, and S.D. Dhodapkar. A translation of statecharts to esterel. In *Proceedings of the World Congress on Formal Methods, FM'99, Volume II, Toulouse, France, September 20-24, 1999*, number 1709 in Lecture Notes in Computer Science (LNCS), pages 983–1007. Springer Verlag, 1999. LNCS nr. 1709.

Table of Contents