HAL
open science

A synchronous model of the PLC programming language ST

Fernando Jiménez-Fraustro, Eric Rutten

To cite this version:

Fernando Jiménez-Fraustro, Eric Rutten. A synchronous model of the PLC programming language ST. 11th Euromicro Conference on Real-Time Systems, ECRTS 1999, Jun 1999, York, United Kingdom. pp.21-24. hal-00546127

HAL Id: hal-00546127

https://hal.science/hal-00546127v1

Submitted on 13 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A synchronous model of the PLC programming language ST

Fernando Jiménez-Fraustro
IRISA / INRIA
F-35042 RENNES, France
Fernando.Jimenez@irisa.fr

Eric Rutten
INRIA Rhône-Alpes
F-38330 MONTBONNOT SAINT MARTIN, France
Eric.Rutten@inrialpes.fr

Abstract

This paper presents first results in the definition of a synchronous model of the PLC programming language ST. This work is part of the integration of the IEC 1131 design standard and the synchronous technology, with the motivation to give access to formal techniques and tools.

Keywords: programmable logic controllers, formal methods, synchronous languages.

1 The design of industrial controllers

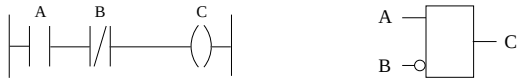
Industrial control systems, e.g. in factory automation, are complex and safety critical systems. Their controlling dangerous or safety critical activities calls for strong requirements regarding the analysis of the correctness of their design and implementation. Certification authorities begin advocating the use of formal and automated methods. They are often implemented on Programmable Logic Controllers (PLC) architectures. Their design relies on standards like the IEC 1131 norm, and more specially its IEC 1131-3 part concerning programming languages [6, 7]. These are various: ladder diagrams, imperative sequential languages like Structured Text (ST), Sequential Function Charts (SFC). This variety reflects that of levels of design and of cultures involved in the design of large controllers, where legacy specifications can be very important. From the point of view of the correctness and safety of controllers, the problem is in the size and complexity of designs. A design assistance has to be provided to designers, giving clear semantics to large specifications, to the way the interoperability of languages works, and a formal basis on which the analysis of specifications can be achieved. The assistance that can be provided by an environment of practical tools to a correct design and implementation is essential. In the area of reactive real-time systems, the emergence of the synchronous technology [5, 9] provides for design environments offering automated support tools for specification, validation (simulation, verification), performance evaluation, distributed implementation, execution on different platforms (HW and OS).

The motivation of our work is to connect the industrial automation design standard IEC 1131 with the synchronous formal model, technology and tools The method followed is based on the modeling of the behavior of the languages of the IEC 1131 norm in terms of SIGNAL. Its implementation in an automated translation opens access to the whole design environment and to its diverse analysis and implementation functionalities. Compared with other work on formalization of PLC languages (e.g. [8, 3, 11]), we want to address the complete norm (i.e. have a formal model of all the languages and their inter-operation), and to connect it to a formally based technology where not only analysis and validation, but also compilation, architecture dependent implementation, code generation are considered. In this paper, we present first results in the definition of a synchronous model of the PLC programming language Structured Text (ST).

2 The IEC 1131 norm and ST

The IEC 1131-3 norm on programming languages is divided in two parts : the common elements, and the four different programming languages. 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 are the basic structuring units, that can be associated to tasks in the execution environment. They can be programs, written in any of the programming languages defined further. Sequential Function Charts (SFC) are a graphical language for modeling the functional and behavioral aspects of discrete-event control systems. Configuration elements give ways to describe the implementation of the controller with the help of global variables, resources, tasks, and acces paths. The PLC programs are mainly structured by means of function

```
LD     A
ANDN   B          C := A AND NOT B
ST     C
```



Instruction List (IL) Structured Text (ST) Ladder Diagrams (LD) Function Block Diagram (FBD)

Figure 1: The four programming languages of the IEC 1131-3 norm.

block units. Function blocks are instances of function block types that encapsulate local data and algorithmic behaviour and respect definite communication interfaces through strongly typed input/output variables. When a function block is activated, input data is passed to its input variables and its algorithm is executed. All the input variables and the local data are accessible for the algorithm. This local data persists from one activation to the next, therefore can be used to record relevant information for future use. After the activation is finished, the caller can read the calculated results from the output variables.

The algorithmic behaviour of a function block is defined in one of the four languages of the norm. 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 behaviour 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. In Figure 1, the four programming languages are illustrated by the description of the same simple program. There are two input variables `A` and `B`, of the Boolean type, and one output variable `C` of the same type; the operation consists in making the conjuction of the value of `A` and that of the negation of `B`. 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. The basic semantics of the languages involves cyclic repetition of a *scan*, where inputs are evaluated, a reaction is computed and outputs are emitted.

3 The synchronous technology

The synchronous approach to real-time and reactive systems [5] offers models and practical design assistance tools with a formal basis. The underlying theory is that of discrete event systems and automata theory. It provides for a technology concretely available as design environments (e.g., the SIGNAL one, or SACRES) with tools for specification (graphical interfaces), simulation, verification, performance evaluation, execution on different platforms (HW and OS).

Some synchronous languages are ESTEREL, LUSTRE SIGNAL, ARGOS and STATECHARTS. The synchrony refers to a particular formal model, with a notion of logical instant, where composability of subsystems is made simpler, and more efficient w.r.t. automated analysis tools. Recently, exchange formalisms, e.g. DC+, allow for the interoperability of the tools of different origins, widening the potential support for languages connected to these formats. Commercial versions of the languages and tools exist, and have applications in industry (nuclear plant control, avionics, ...) [1].

SIGNAL is a data-flow language, where programs are written in the form of systems of equations, composed in a block-diagrammatic fashion [9]; experiments illustrate the ways it can be practically used, and the actual meaning of the synchrony [10]. The basic objects are signals, which are series of values. The programs are equations relating the values of signals at a given instant. The primitive constructs are as follows: functions: e.g. on Booleans: `X := A and not B`; selection (or down-sampling): `X := Y when B`; merge (or up-sampling): `X := Y default Z`; delay (previous value, with initialization): `X := Y$1 init X0`.

They are sufficiently expressive to model finite state machines. They can be composed using the composition operator noted "|". There exist derived constructs for confort and structuring, and external (e.g. C) functions. They can be used to express behaviours 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.

The SACRES project and programming environment is an industrial instantiation of a design environment based on that technology [1]. STATEMATE [4] is integrated to the synchronous technology by providing a translator to SIGNAL and DC+ [2]. 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 [4] features imperative sequential constructs quite close to those of ST [12] ; therefore these results could be re-used here.

4 Modeling Structured Text in SIGNAL

General principle. A function block encapsulates local data that persists from one activation to the next and an 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 statement list in the ST language working on that environment. In imperative languages like ST, every assignment statement changes the environment, and several assignments can change the same variable during the execution.

This section outlines the translation of imperative ST into equational (data-flow) SIGNAL, 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. The case of unbounded loops is mentioned at the end of the section, and principles of a solution are sketched.

The general form of the translation function is $tran(Env_{input}, Env_{output})(stat)$ whith $stat$ the statement to translate, Env_{input} the input environment and Env_{output} the new one, 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.:

$tran(\{\texttt{x1}, \texttt{y1}\}, \{\texttt{x2}, \texttt{y1}\})(\texttt{X:=}expr) =$

```
x2 := tran({x1,y1},{x1, y1})(expr)
```

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

$tran(Env_{input}, Env_{output})(stat;statlist) =$

$$\begin{array}{l} \quad tran(Env_{input}, Env_{stat})(stat) \\ \mid\ tran(Env_{stat}, Env_{output})(statlist) \end{array}$$

At the end of the ST sequence the last environment is saved into the memory for the next *scan*, as follows:

$$\begin{array}{l} \mid\ Env_{current}\ \texttt{:=}\ Env_{endbody}\ \texttt{default}\ Env_{mem} \\ \mid\ Env_{mem}\ \texttt{:=}\ Env_{current}\ \texttt{\$1 init}\ Val_{init} \end{array}$$

where $Env_{endbody}$ is the environment resulting of the translation of the statement list, $Env_{current}$ the one holding the news values and Env_{mem} holding the previous values.

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*:

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

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

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. There is 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. 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 the cases under study. Compared to the other one mentioned next, this approach can be termed *spatial expansion*.

Unbounded 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).

Given the presence of unbounded loops in ST, this technique is necessary to model this kind of behavior.

The mechanism of SIGNAL called upsampling consists in being able to write SIGNAL programs that upsample a signal, i.e. that perform actions at more instants than the occurrences of that signal, i.e. at a relatively faster (or denser) 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 [2, 12].

5 Conclusion and perspectives

The way we approah the integration of the IEC 1131 norm with the synchronous technology is by modeling the languages into SIGNAL equations, supported by an automatic translation. In particular the imperative sequential language Structured Text (ST) is modeled using the data-dependencies encoded as data-flows. 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.

Perspectives are in polishing the modeling of ST (e.g. integrating complex data types), validating and implementing it, modeling other IEC 1131 languages, and taking care of inter-operability questions, making actual use of the synchronous analysis and compilation functionalities, adapted to the specifities of the IEC languages and PLC architectures, and generalizing the use of upsampling to refine a logical instant into a sequential implementation.

References

[1] Ph. Baufreton, H. Granier, X. Méhaut, E. Rutten. The SACRES Approach to Embedded Systems Applied to Aircraft Engine Controllers. In *Proc. of the 22nd IFAC/IFIP Workshop on Real Time Programming, WRTP'97*, Lyon, France, September 15–17, 1997.

[2] J.-R. Beauvais, R. Houdebine, P. Le Guernic, E. Rutten, T. Gautier. 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, 1998 (IEEE Publ.).

[3] A. Fett, G. Egger, P. Pepper. Formal specification of a safe PLC language and its compiler. In *Proc. of the 13th Int. Conf. on Computer Safety, Reliability and Security, SAFECOMP'94*, Anaheim, October 1994.

[4] D. Harel, A. Naamad. The STATEMATE semantics of STATECHARTS. *ACM Trans. on Software Eng. and Methodology*, vol. 5, nr. 4, oct. 1996.

[5] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[6] IEC International Electrotechnical Commission, *International Standard for Programmable Controllers*, IEC 1131 parts 1–5, 1993.

[7] IEC International Electrotechnical Commission, *International Standard for Programmable Controllers: Programming Languages*, IEC 1131 part 3, 1993.

[8] B. Kramer, W. A. Halang. Achieving high integrity of process control software by graphical design and formal verification. *Software Engineering Journal*, Jan. 1992.

[9] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire. Programming Real-Time Applications with SIGNAL. *Another look at real-time programming*, special section of *Proceedings of the IEEE*, 79(9), Sept. 1991.

[10] P. Le Guernic, E. Rutten. Experiments with the synchronous methodology illustrating its support of predictability. In *Proc. of the 21st IFAC/IFIP Workshop on Real Time Programming, WRTP'96*, Gramado, RS, Brazil, November 4–6, 1996. Elsevier.

[11] L. Marcé, P. Le Parc. Defining the semantics of languages for programmable controllers with synchronous processes. *Control Engineering Practice*, vol. 1, nr. 1, february 93.

[12] Mirabelle Nebut. *Modélisation de* STATEMATE *en* SIGNAL *: le langage impératif des actions* Rapport de DEA, IFSIC - Université de Rennes 1, Sept. 1998.