On the Programming of Industrial Computers

Oded Maler

June 4, 1999

Abstract

This report, which is part of the deliverable IP.1 of the Esprit project VHS (Verification of Hybrid Systems), analyzes some software engineering aspects of industrial computers such as PLCs (programmable logic controllers) and DCS (distributed control systems). The report starts with a comparison between the development of software engineering for general-purpose computers and the programming of control computers. Then it critically surveys the five programming languages defined by the IEC 1131-3 standard which is intended to unify PLC programming languages. Finally several potential contributions of the consortium toward improving the state-of-the-art in this domain are suggested.

1 Introduction: Industrial vs. General-purpose Computers

Programming methodologies for the *general-purpose* computer have undergone tremendous improvements since the 50's. In few decades programming moved from machine code and assembly languages, via the first high-level languages, structured programming, data-types, cross-compilers and debuggers up to object libraries, graphical programming environments and inter-application interfaces. The outcome of all these developments (and many others not mentioned here) is that programming can be done at a more abstract and problem-oriented fashion, letting the computer itself do the tedious tasks of handling the technical details of the specific hardware platform on which the program will eventually run. Consequently the cost-effectiveness and quality of software production has increased beyond what could be imagined in the early days of computers.

So far the development of programming methodologies for special purpose industrial computers has been significantly slower.[1] Before we analyze this problem, its origins and, hopefully, some of its solutions, we have to characterize, even roughly, the type of systems we are talking about. First and foremost we are concerned with systems whose major role is to interact via sensors and actuators with a dynamic physical environment, in other words, computers that *control*. Of course, ordinary computers are connected as well to peripherals such as a mouse, a keyboard or a communication port, but this interaction is *not* their raison-d'être. The functionality of control computers is defined in terms of the performance of the physical processes with which they interact. In order to achieve this performance the control system monitors input signals which deliver information from the process, makes some calculations

[1] While the scope of this report is the class of systems controlled by PLCs and DCSs, some of the initial considerations are common to all computer that control, sometimes referred to as "embedded", "reactive", or "real-time" systems.

and produces output signals which are then amplified and transduced into actions that influence the process. The calculations done by the controller are usually simple[2] (compared to arbitrary algorithms which can be performed by general-purpose computers) but they must respect some timing constraints due to the interaction with a *dynamic* environment which does not wait for the controller to conclude its computations.

One can distinguish between two types of signals, continuous and discrete. The former represent quantities such as temperature, velocity or water level while the latter can stand for on/off states of devices such as valves and furnaces, for threshold conditions on continuous quantities or for higher-level supervisory signals. In the pre-computer days calculations over continuous signals where done by analog means, first mechanically and later electrically by analog computers. This is the origin of the block diagram paradigm where input signal move through arithmetical components and integrators to produce output signals. Similarly the discrete "logical" input-output functions were computed initially by electro-magnetic and pneumatic relays and later by hardwired digital electronics, which can be viewed as a discrete version of block diagrams.

The rapid progress in VLSI technology and the arrival of cheap micro-processors affected both types of control systems. For continuous control it turned out to be more cost-effective to replace analog noisy devices by digital calculations. The price of going digital is the replacement of "real" numbers, represented by *magnitudes* of physical quantities, by their binary *encoded* floating-point approximations and of continuous control by sampled piecewise-constant control. The nature of continuous processes (sampling theorems) combined with the speed and accuracy of digital computers guarantee the dominance of digital control. Discretized versions of continuous blocks such as integrators and PID controllers are widely used. On the discrete control side, the replacement of the hardwired (and later programmable) logic controller by a general-purpose microprocessor was even more natural. With a microprocessor inside, one would expect that control computers will converge toward general-purpose computers and benefit from the progress in software engineering but some technical, conceptual and sociological factors are slowing-down this convergence. To analyze the situation let us see what is still special about computers that control.

The major feature of control computers is the need to interact with many I/O signals coming from various types of devices. From the computer hardware point of view this means that the processor board is augmented with specialized I/O boards whose function is to connect physical signals with the computer memory using A/D and D/A conversion, amplification of output signals in order to drive actuators, and communication with the processor. In addition, these computers are supposed sometimes to work in hard physical conditions and be more solid and reliable.

Hence the processor and its software constituted a secondary concern relative to measurement technology and other electrical and mechanical engineering considerations. The intimate connection with the physical world explains the lag in development of more high-level programming culture in this domain. In fact, the situation is similar to that of operating systems programming two decades ago, where "systems" programming was dominated by assembly languages and low-level primitives to handle I/O devices and improve performance.

The diversity of plants to be controlled, such as airplanes and missiles, refineries, CNC machines, production lines at various scales, railway systems, washing machines, alarm sys-

[2]Simple in the technical sense that all instances of the computation can be performed within a-priori bounded time and space. These computations can be very complicated in other senses.

tems, and what not, made it hard to reach a general abstract concept encompassing the common features of all these phenomena.[3] As a result, real-time programming (also known as embedded systems, computer enabled control, etc.) is one of the most eclectic and confused parts of computer science. In some safety-critical domains such as avionics, or nuclear plant monitoring, there was a concentration of a critical mass of software engineers working on the same application using the same hardware and software platform. However, in many industrial domains, software developers work in relative isolation, they are not the dominant sub-community in the enterprise, and their productivity is not the determining factor in hardware selection decisions. Consequently, they are often locked in the specific programming environment supplied by hardware vendors, and need to stay with the same vendor in order to reutilize already developed software. This situation was very convenient for the major hardware vendors.

In contrast, the general-purpose market converged mostly into the *open* architectures of the personal computer or workstation where hardware vendors compete with each other, offering a variety of choices in motherboards, processors, I/O cards, modems, monitors, etc. This open "plug-and-play" architecture encourages competition among vendors, resulting in price reduction and can be viewed as one of the driving forces behind the proliferation of computers in contemporary society. The quest for a similar standard interface is now taking place in the consumer electronics market (IEEE 1394), and will probably change the landscape of this domain as well. As for software, albeit the dominance of Microsoft on the Intel architecture, users have a choice in almost any application domain (including operating systems) and there are numerous programming environments, including several competing compilers for every useful programming language.

Inspired by this state-of-affairs, users and developers of industrial control systems started to push toward an open hardware and software architecture for industrial computers, based on a PC-like computer with a modern programming environment. The advantage of such PC-like systems is their easy integration with other software used by the enterprise. This connection is achieved either by interfacing the PLC operating system with a commonly-used operating system, or by using the concept of a Soft PLC, which is deliever the functionality of a PLC as a software task under an ordinary operating system. At the hardware level, compatibility is to be achieved by the use of one or more out of several standard buses and communication protocols already being used in the industry (some field bus standard already exist for DCS). In order to create a common base for stadards, terminology and software, the IEC (*International Electro-technical Commission*) formed several technical committees resulting in the so-called IEC 1131-3 standard, published in 1993.[4] Later, an independent association called PLCopen was created to promote the usage and supply of products in conformance with this standard.

[3] Compare again with the much simpler domain of business data-processing, in which it took many years to develop the unifying concept of the data-base management system, separating the logical from the physical in information storage.

[4] This is part of a more general standard, supposed to define common terminology in the chemical process control world, but we will concentrate on the software.

2 The IEC 1131-3 Standard

2.1 Introduction

The IEC 1131-3 standard (hereafter "the standard") is an attempt to unify, at least at the syntactic level, the main types of languages used in practice for PLC programming around the world. Before getting into the details, some general comments about *theory* and *practice* are in order. From a theoretical point of view it is somewhat strange to read a document that speaks with the same importance about details such as character sets, and about how to connect and evaluate function blocks: the former being a theoretically-trivial question while the latter is a deep semantic issue on which numerous papers have been written. Neither does a theoretician feel comfortable with a document starting as a legal contract with a list of 79 terms and their meaning in the text. These terms are of various sorts including *absolute time*, *bistable function block*, *generic data-type*, *resource*, *task*, *carriage return* and ... *semantics*. These sentiments are not particular to the IEC standard and one can feel the same toward formalisms such as VHDL (hardware definition language), SDL (a language for specifying distributed systems) and UML, which is supposed to be a "universal" modeling formalism, supported by software industry giants. All these formalisms seem to put too much attention to notation and features (to satisfy all committee members), while neglecting the semantics, that is, what is the meaning of the specification or the program written in them. However, let us not forget that:

- Theoretically-trivial questions such as compatibility of character sets (or electronic plugs, for that matter) are prerequisites for *any* possibility of connecting devices and software units together.
- The engineer needs to solve problems in real-time and produce solutions for concrete problems today. The esthetics, generality and scalability of the solutions are of secondary importance. People started to communicate with each other long before having any abstract ideas about grammar or meaning.
- Theoreticians have a tendency to want to start everything from scratch. However in real life, backward compatibility is very important, even if the price is carrying with you some of the anachronisms of previous solutions. Moreover, practitioners tend to hold on to formalisms and tools they are used to (see Fortran or Cobol) as long as they feel they solve their problems.

Having this in mind, together with the fact that before the standard, many PLC vendors provided more or less the same functionalities but with different syntax, one can appreciate the enormous progress which the standard has brought, although some aspects of the standard will be criticized from a theoretical standpoint in the sequel. Five classes of languages are covered by the standard:

1. Instruction List (IL): An assembly language inspired by languages used in various existing hardware platforms.
2. Structured Text (ST): A Pascal-like imperative general-purpose programming language.
3. Ladder Diagram (LD): Essentially the popular graphical RLL formalism used mostly in the US. Ladder diagrams are essentially yet another way to write Boolean switching functions based on the metaphpor of relays.

4. Function Block Diagrams (FBD): a data-flow formalism for describing a network of function blocks connected by signals.

5. Sequential Function Charts (SFC): A graphical formalism inspired by the French (standard) formalism Grafcet, based on a variation of a class of Petri nets. This formalism allows a combination of sequential and parallel activities and is popular in Europe. The standard is adopted mostly from an older standard called IEC 848 defined in 1988.

As one can see, the languages come from various origins. IL and ST are in the computer science tradition of sequential programming, while LD and FBD imitate the structure of their predecessors, that is, hardware implementation of discrete and continuous controllers, which are essentially parallel by nature. SFC combines sequentiality with parallelism.

The standard does not try to define compatibility relations between these formalisms. It allows (without a guarantee of meaningfulness) to combine elements from different languages. Many lexical, syntactical and graphical conventions are shared by all these languages.

2.2 Common Features

The standard starts with a description of the hardware and systems software environments (*configuration* and *resource*) on which programs are supposed to be run. The simplest and most generic case is a single control program running on single PLC. Such a program reads its input, calculates its state and writes its output. The details of how a specific PLC performs I/O and memory management operation are not part of the standard, which is based on a layered architecture. It is assumed that the PLC sensor readings and actuator values are passed through machine-specific memory locations, and the application software need not be concerned with these details. This part of the standard also links the programming standard with higher-level parts of the IEC-1131 standard for which PLC systems are among the building blocks. The standard has constructs for allowing different programs to be loaded and run on the same PLC (separately or under multi-tasking) and for several PLCs running in parallel and communicate via "access paths" which are abstractions of communication protocols.

A *program* is built from a number of different software elements, written in any of the five languages (typically function blocks), which may exchange data among themselves. These software elements are composed in parallel and are not invoked by themselves unless they are assigned to a *task* and the task is either triggered by an event or configured to execute periodically.

All languages share the same character sets and conform to ISO standards and conventions for encoding time stamps. The standard defines elementary data-types, declaration of compound data-types, initializations, etc. Variables can have local or global scopes. This is standard stuff in modern programming. In addition to the usual abstract variables, there are "directly represented" variables which are addresses in the input, output and internal memory locations. There is also a distinction between normal and "retainable" variables, where the latter are supposed to keep their value after a physical shutdown of the computer.

2.3 Functions and Function Blocks

Function blocks are one of the basic elements of the standard, a special case of which are the (memoryless) functions which we discuss first. These are functions that have no internal

variables that persist between two invocation, and hence produce the same output for the same input each time they are called. Basic built-in functions can be composed together in an acyclic fashion to yield new functions. This is a well-known and non-problematic practice in sequential programming and in the design of combinatorial circuits.

Functions can be written either in the textual ST language (standard Pascal-like definition) or in a graphical formalism used for the FBD language. The syntax of the latter is defined using a mixture of text and ASCII graphics whose origin is probably related to backward compatibility with existing programming environments. Someone more aware to the distinction between syntax and semantics and between internal and graphical representation, would have probably offered a cleaner formulation based on an abstract mathematical representation of a network of functions (along with its isomorphic internal computer representation), a compilation of textual programs into this format, and a graphic editor which can extract the structure of the network from a user-drawn graphical layout. This is in fact what is done today by IEC-1131-3-based tools.

The standard offers numerous built-in functions including, type conversions, numerical operations, boolean functions and string manipulation and selection functions. In general, functions do not seem to pose any serious semantical problems as long as combinatorial loops are avoided.

Function blocks, which are function with memory, constitute the major software element of the standard. In software terminology, a function block is a *reactive module* with its own variables and data-structure and an interface with the outside world.

Here, we believe, a better understanding of the theoretical issues involved, would have improved the standard. The objects described by function blocks are as well functions, but not functions on "static" data-types such as reals, or integers or Booleans or some aggregations of those, but rather functions on *sequences* of elements taken from these domains. For example, the function block DEBOUNCE appearing in Figure 10 of the standard is *not* a function from BOOL $\times$ TIME to BOOL $\times$ TIME but a functions that maps sequences of BOOL $\times$ TIME to other sequences of this domain. In theoretical terminology this is a sequential function or a transducer, which can be represented by an automaton or by a circuit with latches. Of course, memoryless functions, described in the previous paragraph, such as AND which maps pairs of Booleans to Booleans can be extended naturally to functions on sequences, but since at each time instance the current output depends *only* on the current input, this point of view does not contribute much. For functions with memory, this insight is indispensable, and is in the heart of the distinction between transformational and reactive systems.[5]

Function block declaration are syntactically similar to functions, except for having internal variables which can be updated at every invocation and retain their value after each invocation (this is an indirect way to speak of sequences). In fact, the declaration of a function block is viewed in the standard as declaring a type, and then instances of this type are declared as variables. In principle function blocks can be transferred as arguments to function blocks, which is a semantic can of worms, I am not sure the authors would like to open.

There are many standard function blocks such as flip-flops and counters. Other function blocks provide discretized versions of continuous-time operators such as integral and derivatives. Integration, for example, is done at each cycle by adding the product of the input and

[5]These claims should not be interpreted as preference of I/O descriptions over state-space descriptions. The only message here is that what function blocks do is to transform input sequences to output sequences. A representation by a program or ab automaton can be as good and sometimes preferrable to an I/O description, as long as we remeber what is the functionality of the object in question.

the size of the time step. Here, again, a careful understanding of the objects in question (in this case, discrete time and continuous time signals and functions defined on them) could contribute to clarifying the text.[6] As an example of blurring this distinctions, we can look at section 2.5.2.3.4, where definitions of standard blocks called *timers* are given. They appear only graphically without their definition in textual language. Moreover, for some of them it is specified explicitly that they cannot be used in textual languages. Their "semantics" is exemplified in table 38 using what appear to be continuous time signals.

2.4 The Languages

2.4.1 The Textual Languages IL and ST

The textual languages are *Instruction List – IL* and *Structured Text – ST*. Both are essentially classical sequential and imperative languages, the first being a low-level assembly language and the second a high-level Pascal-like language. Such languages are fairly standard in computer science and we have only the following comments:

- Using the full expressive power of these languages (e.g. WHILE loops) it is possible to write procedures whose execution time is not predicted, not bounded and even infinite. Incorporating such programs in control application is, of course, not a healthy practice. On the other hand, it is possible to impose syntactic restrictions which may guarantee bounded response time. For example, one can allow only programs with no backward jumps (in IL) or restrict FOR-loops to have constant delimiters (in ST) to guarantee this property.
- These languages are inherently *sequential* which makes them insufficient for being a formalism for writing whole control applications, which almost always have parallelism. In the absence of an explicit parallel composition construct, concurrency is achieved either using additional multi-tasking definitions (as in ADA), or doing the parallel composition via other languages (FB and SFC) which can accept sequential modules as building blocks. An alternative approach is demonstrated by the real-time imperative language Esterel, or by formalisms such as CCS and CSP, which do admit an explicit parallel composition operator in the syntax of the programming language itself.

In any case, these languages are well-suited for writing modules by programmers having a general computer science (rather than control) culture. The IL language can serve as a basis for an abstract machine to which other language can compile.

2.4.2 Ladder Diagrams – LD

Ladder diagrams is a graphical language designed for backward compatibility with the RLL formalism, itself a result of backward compatibility with hardware relay technology. Essentially what you want to write in LD are relations between the values of current state and input variable and the values of next state and output variables, plus some suggestions on the order of evaluation of the conditions involved. The technology of Boolean expressions and Binary

[6] In MatrixX/Xmath, a popular block diagram package for control engineers, the delay operator z for discrete time signals is distinct from the delay operator on sequences which is called there "shift register" although their functionality is the same.

Decision Diagrams (BDD) seems to me much more suitable for this purpose then the language of relays. In fact, relays resemble transistors, which are semantically more complicated than Boolean gates. The use of transistors is justified in the design of digital circuits where they are closer to the physical implementation medium than their Boolean abstraction. But as a metaphor for decision making, which is later to be compiled into software, there is no real reason to use the anachronistic relay metaphor, which can be encoded using Boolean block diagrams. Of course, this is a subjective opinion and the death of this dialect can be a very slow process...

2.4.3 Function Block Diagrams – FBD

Function block diagram (FBD) is a graphical language for composing simple function blocks together to form larger ones. The interaction between function blocks is represented by "wires" connecting output variables of one block with an input variable of another block. A composition of several blocks can be encapsulated into one big block, encouraging a modular and hierarchical style of program development. There are, however, certain programming constructs which are not comfortably expressible using graphical notation. These include FOR-loops and operations on arrays, interrupts which lead to abortion and complex algorithms in general.

Function blocks diagrams resemble very much the data-flow language *Lustre* which underlies the programming environment *Scade*, used in avionics and nuclear plant control. Due to the safety-critical aspects of these applications, Lustre is based on a very precise semantics (functional equations on sequences) and goes through a compilation process which includes checking whether the program is well-defined (no causal loops) and a generation of an optimized C code which runs all the program as a single loop (no multi-tasking). Although many applications of PLC systems are not as safety-critical nor time-critical as flight control,[7] we believe that some of the insights gained in the Lustre experience, such as simple sequence-based semantics, few primitives or explicit delay operator, can contribute to the development of future versions of FBD and their corresponding semantics. On the other hand, the IEC idea of allowing certain blocks to be written in a well-behaving subset of an imperative language, directly in the programming environment (unlike connection to C routine in the linking phase) might be useful as an extension to Lustre in cases where the data-flow formalism is not adequate.

2.4.4 Sequential Function Charts – SFC

Sequential function charts constitute a formalism which combines sequential and parallel operations. Whether is should be considered as a fifth IEC language or as more high-level structuring tool is a question of terminology. SFC is based on Grafcet which can be roughly characterized as a synchronous and labeled variant of Petri nets. There are many incompatible interpretations of this important formalism (again, the standard is rigorous about the orientation of connecting lines, but less so concerning the operational semantics) and we will try to give the main principles.

A basic entity in SFC is the *step*. In fact, it is not easy to understand and explain this notion without having a clear distinction between the state of the PLC system and that of

[7] On some popular packages for PLC programming, such programs are run via an interpreter, and certain applications are slow enough to run even on Windows NT!

the environment. Roughly, from the point of view of the PLC, a step is part of its state (when there is no parallelism, the step is the state). When a PLC program is in a step, it typically implies that certain output variables controlled by the program (what is called "actions", see below) are kept in a certain value. For example, a step "heat" in a PLC program might mean that a a certain Boolean variable, whose value is actuated into the heating device, is in a state ON. The actual physical process which underlies the step might be more complicated and include lower-level feed-back loops, but at the level of the SFC it is represented by one or more variables which stay constant during the period in which the step is active. This is similar, to a certain extent, to the layered architecture used in communication networks, where what is viewed as a "transmit file" step at one layer is realized by a complex dynamic process in a lower layer. Steps are represented graphically by rectangular boxes.

Two consecutive steps are separated by a *transition*, which is essentially a condition over input variables (the condition can be written using various IEC languages, but this is not the important point). When the transition condition is true, the first step terminates and the next state starts. In the heating example, a condition might be "temperature more than 30" which refers to a sensor reading of a variable influenced by the step (of course, from the PLC point of view the two variables are unrelated – it is only through the physical environment that they become related). A transition condition can be any other external event triggered, for example, by the operator. The termination of the first step is accompanied by "undoing" some of what has been done by the first step, for example, turning the heater OFF. What is reset and what is retained depends on the *qualifiers* of the *actions* which constitute the step – see later. Graphically, a transition is a bold horizontal line crossing the vertical line connecting the two steps (see Figure 1-a).

There are two special variables associated with every step. One is a Boolean variable indicating whether the step is active or not (whether it has a token, in the Petri net terminology). The other is a timer which measures the time elapsed since activation. This variable can appear in transitions like any other variable and allows to specify time-bounded behavior such as "heat for 5 minutes".

So far we have described "straight-line programs" without choice. The mechanism to implement choice is to use divergent paths, that is, to split the line leaving a step into two or more lines, each with its associated transition condition and next step. The conditions need not be mutually exclusive and they are evaluated using a default or a user-defined order to decide which branch will be taken.[8] The notation is somewhat unfortunate because the bifurcation of the lines takes place before the competing conditions and it may lead to some confusion with parallelism.

The parallel composition operator is represented graphically by a horizontal double line, from which several parallel sequences can emanate. In that case the state of the system is the set of the states of the parallel processes which proceed independently until they merge again. Such a "synchronization" is represented by another horizontal double line to which all the last steps of the involved processes converge, and the transition following that line terminates these steps (see Figure 1-c).

Using parallelism it is very easy to produce bugs and meaningless programs. One possibility is to modify the same variable in two or more concurrent branches of the program. Another possibility is to "synchronize" two branches which are exclusive (not concurrent).

[8]Of course, it is theoretically trivial to convert an ordered set of conditions into an equivalent unordered and mutually exclusive set, but some users might prefer this ELSE..IF construct.

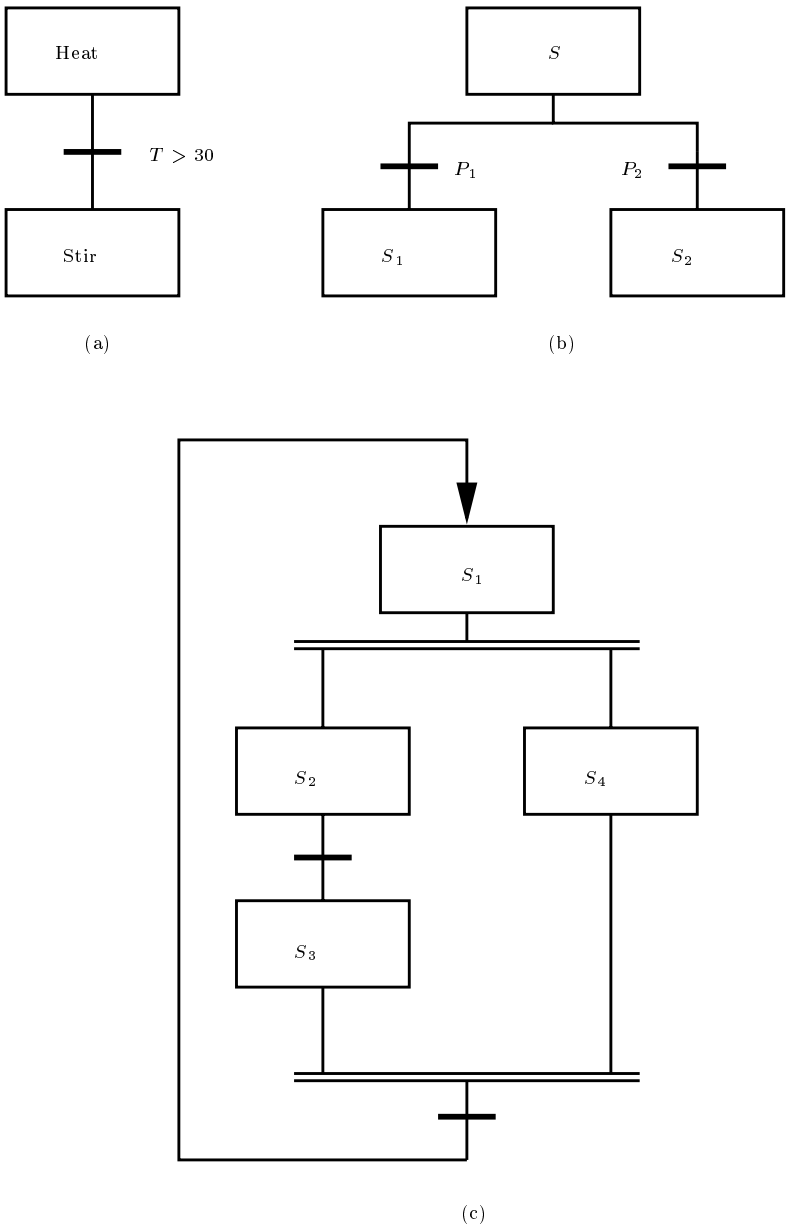

Figure 1: Examples of SFC constructs: (a) sequence; (b) choice and (c) parallelism.

This will cause a deadlock. Other forms of unsafe programming might create an unbounded number of parallel steps. Most of these problems can be avoided by a discipline of programming which restricts the syntax of programs which are accepted by the program development environment.

So far we have avoided a discussion of the semantics of SFCs. At the theoretical level, the appropriate objects are continuous-time signals, most of which are discrete-valued and the rest are clocks. Assuming a non-Zeno behavior (values of external variables and conditions do not change infinitely many times in a finite interval), SFC programs can be viewed as signal transducers. In order to avoid ambiguity, one can assume that no two events happen at the same time, and that every condition is fired as soon as it is true. This ideal semantics is approximated by an implementation where the input variables are sampled periodically. Here two events can happen at the same cycle, one making a condition true and the other falsifying it. The interpretation rules of Grafcet are supposed to give an unambiguous semantics to such cases.

As mentioned above, steps can be contain a sequence of actions which can have various qualifiers determining the duration of an action during the step lifetime. Some actions can be active during the whole step, some can be "done"[9] only at the beginning or the end of a step, some may be delayed, etc. Personally, I feel this could be done more elegantly using fewer types of action qualifiers and more steps. Some implementations allow hierarchical design where an SFC is regarded as a step by a higher-level SFC and there is no reason why steps with multiple action qualifiers cannot be broken into sequences of steps.

To summarize, SFC is a powerful formalism which seems to be natural for processes which combine sequential and parallel aspects. I cannot avoid remarking that in theoretical computer science, Petri nets are considered part of the theory of *concurrency* while from the control point of view they are viewed as *sequential*, and indeed they are, compared to block diagrams.

3 Potential Contributions of the Consortium

The contribution of the reactive systems community[10] to the enterprise of programming industrial computers can be in the following inter-related domains:

1. Giving a precise semantics to PLC programs and their physical environments. Concurrent and distributed systems have been investigated by computer scientists for years and exposing the useful essence[11] of this knowledge (mutual exclusion, synchrony vs. asynchrony, causality and so on) can do good for both communities. More recent research on timed and hybrid systems may clarify subtle issues concerning the interaction of the computer with its environment.

2. Influencing the development of design methodologies, language standards and programming environment toward the more rigorous side. This effort should be based on the accumulated experience of developing and studying various languages and tools (such

[9] There is some confusion already in the name "action": is *keeping* the heater ON an action in the same sense that *incrementing* a counter is?

[10] This broad term refers to computer scientists working on the semantics, verification and programming methodologies for computers that interact with an external environment.

[11] That is, around 1% of the publications.

as Lustre/Scade, StateCharts and Esterel) for other application domains. General computer science know-how, such as compilation technology, which is not specific to the reactive systems community can be useful as well.

The question of whether such a contribution is possible via interaction with users, technical committees, hardware or software vendors is an empirical one. Hopefully it will be answered by the end of the project.

3. Development of verification technology for PLC programming. Controllers written in a well-defined language can be subject to formal verification which is equivalent to exhaustive testing of the program in front of *all* admissible behaviors of the external environment. For program properties which do not require modeling of the environment, "classical" discrete verification is already applicable, as witnessed by some work on case-studies 1 and 2. Transforming programs written in well-behaving subsets of the IEC languages into formats used by existing verification tools is a standard exercise.

   For time-dependent properties, the new technology of timed automata (Kronos, Uppaal) can be applied, although a lot is still to be done in terms of modeling principles and more efficient verification algorithms. More intricate properties require modeling of the environment. Whether a heating step, whose termination condition is that the temperature passes a certain threshold, indeed terminates, depends on the fact that the temperature is monotonically increasing and diverging (at least in a certain range) when heat is on. More detailed properties, such as quantitative estimation of step durations, require finer levels of modeling. Finding the most abstract description level of the external physical dynamics which is still sufficient for verification of interesting properties is a major challenge for the rest of the project.

4 Conclusions

The programming of industrial computers is still shaped by languages used for old technology and backward compatibility, but the first signs of an evolution toward a more structured and high-level discipline of programming are already visible.

Acknowledgements: The views of the author on the implementation of controllers and real-time programs were mostly shaped by numerous discussions with Paul Caspi. Stefan Kowalewski helped in understanding some IEC-1131-3 concepts. Discussions and exchanges with S. Engell, J.-M. Flaus, J. Camand, P. Niebert, Y. Giroud and E. van der Wal contributed to increase the correspondence betwee this report and reality. Excessive opinions and factual inaccuracy should be attributed to the author.