The Argos Language: Graphical Representation of Automata and Description of Reactive Systems

F. Maraninchi
Laboratoire de Génie Informatique
Institut d'Informatique et de Mathématiques Appliquées de Grenoble (IMAG), France

Appeared in the proceedings of the IEEE Workshop on Visual Languages, Kobe, Japan, October 1991

Abstract

We present the Argos graphical synchronous language for the description of reactive systems, and the Argonaute environment associated with it. Systems like communication protocols, real time process controllers or man/machine interfaces contain a reactive kernel. *Its behaviour can be described in a convenient manner by an automaton , for formal validation purposes. But, in general, complex systems cannot be described directly as automata. The Statecharts [4,5] and Argos [7,8] are automata-based languages. The high level constructs of the language deal with states and transitions directly. A consequence of this choice is the* graphical *syntax, since the best representation of small automata is graphical. A consequence of this graphical syntax is the need for* graphical *constructs: the constructs of the language must allow the decomposition of a system into small parts that can be represented directly by automata, and they must be given a readable graphical syntax.*

1 Introduction

The term *reactive* was introduced by A. Pnueli [6] to qualify the class of systems like communication protocols, real time process controllers, man/machine interfaces,... Specific problems arise for these systems, because of their intrinsic complexity. Moreover, powerful description techniques, high level languages, adequate representation techniques or validation tools do not exist yet.

Reactive systems are opposed to *transformational* ones, like compilers, which can be described in an appropriate manner by giving the output as a function of the input. There exist methods that allow to decompose the behaviour of such a system into smaller parts, and there exist languages which support these methods. (functional languages, with the composition of function as high level construct, or imperative languages, with the notion of procedure).

For reactive systems, this is not the case, because this is not very clear yet what decomposition methods can be applied, and what language constructs can be introduced to support them. Indeed, a reactive system maintains a continuous interaction with an *environment*, and its behaviour is a set of sequences of elementary interactions between the system and the *environment*. Reactive bahaviours are intrinsically parallel, because the system is considered as evolving in parallel with its environment, taking input from it, and sending output to it.

However, a lot of work is being done on the topic of reactive systems, and there exist several languages for the description of such systems, together with programming environments. We are particularly interested in programming languages, like Esterel [1], Lustre [2], the Statecharts [4,5] or Argos [7,8], whose semantics rely more or less on the same model. This model is the notion of automaton, or labeled transition system, which is very well adapted to the representation of reactive behaviours.

A system like the digital watch is described in a very convenient manner by seing the running modes of the watch as *states*, and the changing of modes as *transitions*. States can be `Watch`, `Stopwatch`, `Setwatch` and `Alarm`, and the user may change modes by depressing buttons *Upper Left* or *Lower Left*. This constitutes the *physical* events. We associate to them the *logical* events `UL` and `LL`, to be used in the description of the watch, as the labeling of transitions. The description of a reactive system is the description of its reactive *kernel*, which deals with logical events. The interface between the environment and the reactive kernel, which translates physical input events into logical

ones — and logical output events into physical output events, has to be described in another language.

We are particularly interested in the use of automata for the description of reactive kernels. Following this idea, they are mainly two approaches that can be adopted in order to design a high level language for reactive systems.

The state-transition paradigm can be considered to be too low level. It is used only at the model level, and the language constructs are chosen according to another way of designing the reactive systems. This is the case for Esterel and Lustre. The designer does not "think" in terms of automata.

The other approach is to consider that the state-transition paradigm is powerful enough, in the case of reactive systems, to be the basis of a high level language. This means that the user will have to "think" in terms of automata. The limits of this approach are reached quickly: the design of a complex reactive system, as a single automaton, is completely unrealistic. A way of *structuring* automata descriptions has to be found.

The language we present here follows the second approach: the state-transition paradigm is considered to be powerful enough to be the basis of a high level language. Moreover, it claims that the best representation for automata, provided they are reasonably small, is *graphical*. So the "semantical" requirements and the "graphical" one converge: the language must provide the user with high level constructs to support design methods, so that the basic components of a program are always small automata; and these constructs must be given a "good" graphical syntax too.

The Argos syntax is originally inspired from that of Statecharts, but the language constructs are different. To the author's opinion, Statecharts make a more extensive use of the graphical syntax than Argos. For small systems, the Statecharts representation may be more concise than the Argos one, and, in this sense, more readable.

But concision is not the only criterion for readability, especially when the size of the systems we consider grows. Even if a picture is worth a thousand words, there will always be systems big and complex enough that cannot be represented on one physical paper sheet. So the language must provide a way to cut the representation into smaller parts. The point is that this operation must not be only syntactical. Cutting a representation *a posteriori*, without taking into account the semantics of the system described will make it unreadable (the same is true for textual languages: listings are cut between procedures, or between control structures, if procedures are too long). The representation must have a graphical structure, in order to be cuttable, and this structure must be the semantical structure of the system, for the different parts to be meaningful.

In the Statecharts, this is clearly not possible, because there is no way the representation of a complex system can be cut into small graphical parts which represent sub-systems (see section 3.3, which gives some hints to understand this). A system has to be designed globally, and represented globally.

Argos proposes a different use of the graphical representation of automata. A program is either an automaton, or the result of applying a unary, binary or n-ary operator to program operands. Each automaton is drawn with boxes and arrows. Each operator is given a graphical syntax and the representation of operators constitutes the squeletton of the program: its graphical and semantical structure.

2 Representation of simple reactive behaviours

A simple reactive behaviour may be described by a *labeled transition system* as shown by figure 1. The transition system has one *initial* state. Transition labels are made of two parts: the *input* part I, and the *output* part O. The complete label is denoted by I/O. Both parts are built upon a set `E` of elementary interactions with the environment, called *events*. The input part is a conjunction of events or negations of events. It describe a condition to be fullfilled by the environment in order to make the system react. The output part gives the events the system outputs to its environment, when reacting to a given input. In the sequel, we shall refer to the buttons of the watch as `LL`, `LR`, `UL` and `UR`, standing for: *Lower Left*, *Lower Right*, *Upper Left* and *Upper Right*. We introduce logical output events to control the physical state of some lamps, used to show the current running mode of the watch (in a more realistic description of the watch, the changing of modes makes the display of the watch change, but it is not reduced to "lamps"; see below). The `X` lamp may be switched on and off with events `X_on` and `X_off` respectively. The `Watch` lamp is initially on.

The first advantage of the graphical representation is that it makes the detection of non-determinism very easy; it is an important notion for reactive systems. In spite of the inherent non-determinism in the description of the environment, the programs should de-

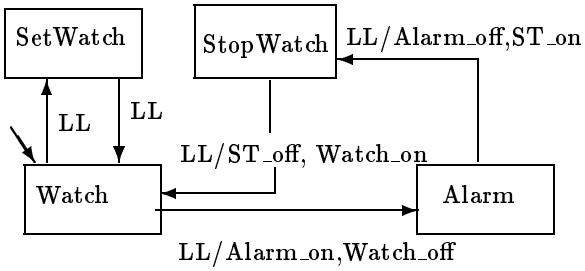

Figure 1: A reactive behaviour

scribe deterministic behaviours. In this framework, non-determinism of a reactive behaviour is simply the existence of two transitions sourced in the same state, with the same input part, and different output parts and/or target states. In the textual representation of an automaton, the labels of the transitions sourced in a given state are not necessarily grouped, and conflicts like non-determinism may be difficult to read. In the example of figure 1, the reaction of the watch to button `LL` when in `Watch` mode is not deterministic. A single button cannot be used for two functions, and one `LL` input of the two transitions sourced in `Watch` mode should be replaced by a new logical event, related to a new button (see figure 2).

3 The Argos constructs

This section constitutes a very brief presentation of the language constructs, focusing on the relation between semantical and graphical constraints. It is necessarily incomplete concerning the precise semantical definition of the language. However, the following points can give sufficient hints for the understanding of the paper. The graphical representation is only a *syntax*, with meaningful aspects (i.e. connections between boxes and arrows) and meaningless ones (i.e. size of boxes). There exists an equivalent textual syntax. There exists a unique underlying model (the compiled form of a program) for each valid graphical representation. There exists a graphical representation for each valid model, since this is an automaton, and all automata can be represented in the Argos syntax. There exists several graphical representations for any valid model; they may differ on meaningless aspects (as two identical textual programs may have texts which differ on the number of blank lines), or on meaningful ones (the compiled form of a program may be obtained by using different constructs at the language level).

3.1 Parallel composition and local events

We consider now a more realistic digital watch. Observe figure 2. We describe the watch as a set of *parallel* components, which communicate with each other. The *interface* we described partially is completed, and we add special components in order to control the display of the watch. Components are separated by dashed lines. In this example, they are all automata, but they could be composed systems as well. The parallel *operator* is defined formally as a binary operator, but it is commutative and associative, so the graphical syntax illustrated by the figure makes sense: there is no need for explicit parenthesis when there are more than two components. Associativity and commutativity of the parallel composition is a rather strong semantical constraint, but explicit parenthesis (for instance with rectangles to group components together) would make the graphical syntax too complex.

The interface deals with input events (the four buttons) and contains exactly the information which is necessary to interpret them correctly. The interface is the only component of the system which "*knows*" that `LL` when in `Watch` mode means "*change mode*", `UL` when in `Watch` or `Alarm` mode means "*enter the corresponding set mode*", `UR` means "*toggle alarm indicator*", but only when in `Alarm` mode, etc. Other components do not deal with input events directly. The interface performs a translation from the logical events which correspond to buttons, to events like `comAL` (for "*commute alarm indicator*") or `comMODE` (for "*commute mode*"). Conversely, the interface does not deal with the display of the watch. Other components do.

The *Alarm indicator* component is used to memorize the state of the alarm indicator. When changing states on `comAL`, it switches on or off a little lamp of the watch display, which can, for instance, display a little bell when alarm is on. This lamp is initially off. The *Main display* component maintains the state of the main numeric display. It can show either Hours, Minutes and Seconds, or Hours and Minutes only, or Minutes, Seconds and milliseconds.

Communication between components is done by the events which are output by one component and input by another one, like `comAL`. The semantics of the communication mechanism is defined formally as a *synchronous broadcast* [8]. When a component outputs an event, it broadcasts it towards its whole environment (the other components, and the global environment of the system). Similarly, its inputs may be inputs from the global environment, or events output by another component.

The description of a reactive system often intro-

duces a lot of events which are used only for internal communication. Argos provides the designer with a way to declare *local events* (see figure 2). The rectangular box labeled with `comAL, comMODE` defines the scope of events `comAL, comMODE`: inside the box, these events can be used as inputs or outputs between components. But, when used as input, they cannot come from the environment outside the box, and when output, they are not visible outside the box.

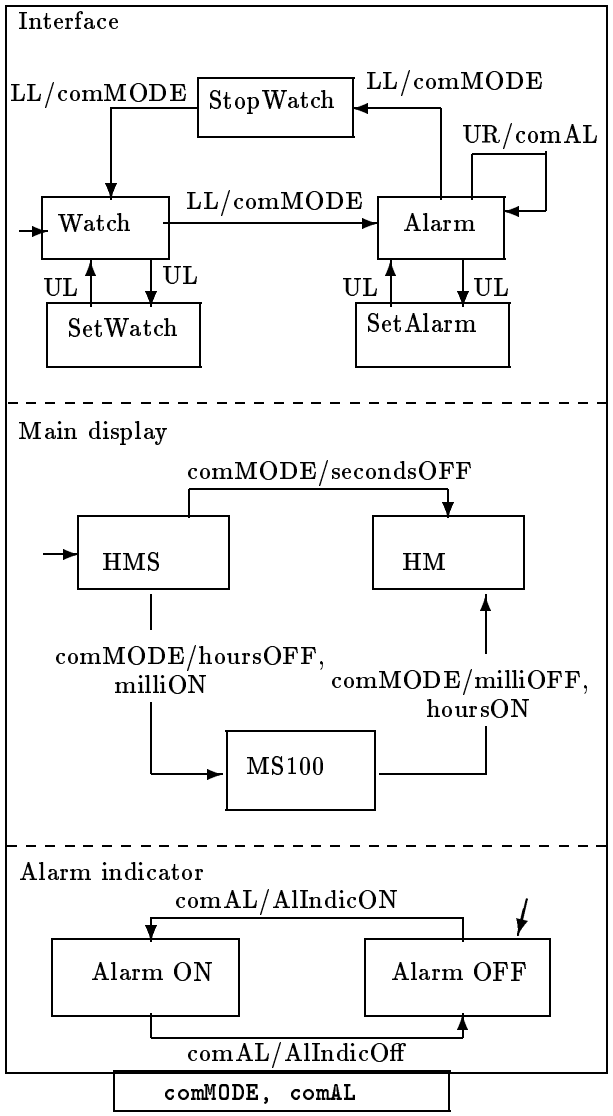

Figure 2: interface and ressources of the watch, an example of parallel components

3.2 Refinement operation

We focus on the interface component now. As described above, it does not express all the meaning of the buttons. Figure 3 shows a complete description of the interface. We introduce the following events: `comLAP` and `comRUN` to control the stopwatch behaviour, `comCH` to commute chime mode (when chime is active, the watch beeps each hour), `incrx_H` and `incrx_A` to control the time keeper component.

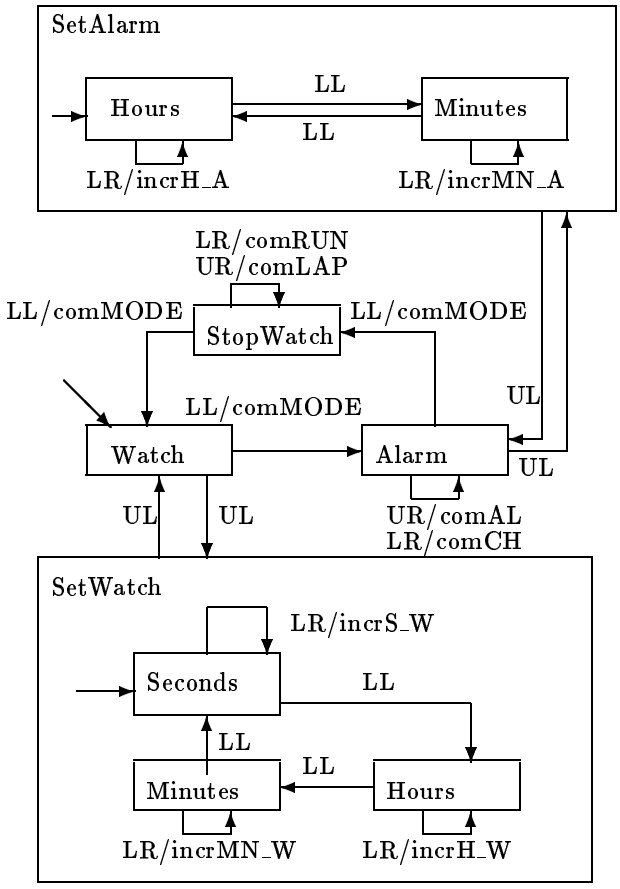

Figure 3: interface of the watch, an example of refinement

The `SetWatch` and the `SetAlarm` states are *refined* by sub-systems. Observe the `SetWatch` state. When the watch is in `Watch` mode, and the users presses `UL`, the `SetWatch` state is entered, and the sub-system which refines it is *started*, in its initial state, i.e. `Seconds`. Then, two sub-systems are active:

- the *controller* with states `Watch, Alarm, SetWatch, SetAlarm` and `StopWatch`, which can react to `UL`
- the *controlled* sub-system with states `Seconds, Hours` and `Minutes` which can react to `LL` or `LR`. `LL` is used to change fields, and `LR` is used to increment a field. The incrementation outputs appropriate events to a time keeper component not represented here.

When the controller reacts to `UL`, the controlled subsystem is *killed*, and the watch reenters `Watch` mode.

3.3 Comparison with Statecharts

Due to its graphical syntax, Argos may seem to be very similar to Statecharts [4], one of the possible applications of the notion of Higraph presented in [5]. This paper gives a lot of ideas about the constructs that can be introduced in a graphical syntax in order to improve the representation of a big and complex state-transition diagram. It introduces the notions of orthogonality (cartesian product) and hierarchy (ability to cluster states into a macro-state).

When applied to the description of reactive systems, these ideas give Statecharts. They lead quite naturally to design methods and programming style which are very different from that of Argos. Indeed, Statecharts are very well adapted to the decomposition of the state space of the system, as an AND/OR tree. AND nodes correspond to orthogonality, and OR nodes to hierarchy. The intuitive meaning is the following: when the system is in an OR state, it is in one of it sub-states; when it is in an AND state, it is in all its sub-states at the same time. Figure 4 illustrates the state space structure for a version of the watch where the behaviour of the `StopWatch` state is detailed. Two examples of authorized arrows are given in the Statechart. The *configurations* (global states) of the watch are: `Watch`, `H`, `M`, `S`, `Alarm`, `SetAlarm`, `lapon` $\times$ `runon`, `lapon` $\times$ `runoff`, `lapoff` $\times$ `runon` and `lapoff` $\times$ `runoff`.

The behaviour of the watch, which is a set of transitions between configurations, is expressed by drawing arrows between boxes of the state space representation. Statecharts allow arrows between any two boxes of the representation, even at different levels. The upper dashed arrow of figure 4 is a transition between configurations `lapoff` $\times$ `runoff` and `Watch`. The lower dashed arrow is a a *set* of transitions: `lapon` $\times$ `runon` to `M`, and `lapoff` $\times$ `runon` to `M`.

This is the feature which makes the Statecharts more concise than Argos sometimes. But this is also the reason why there is no way to distinguish subsystems in a big Statechart, and, consequently, it is very difficult to cut the figure into meaningful parts. In some sense, *inter-level* transitions can be compared to GOTOs in classical sequential languages.

4 The Argonaute graphical environment

Argos is currently used as the basis for the Argonaute verification environment. It provides the user with a complete graphical and interactive environment, in which he can edit Argos processes, visualize them, simulate their behaviour, etc. Processes are saved in text files, using the ARGOS format, which contains both semantical and graphical information. These files can be obtained from a text editor, too. They are taken as input by the *Argos compiler*, which produces a labeled transition system, according to various formats. Argonaute is connected to "semantical" tools like a automaton comparator, used for the verification purpose (Aldebaran [3], developed at IMAG-LGI).

The current release of Argonaute runs under Xwindow 10 [1] on Sun3 Stations. The next release is under development and will run under XWindow 11 on Sun3 or Sun4 stations. It is written in C language.

The graphical editor is syntax-directed. The user build automata first, by creating states and naming them, creating transitions and labeling them, designating an initial state and so on. Then he can build composed systems by choosing an operator and designating its operand(s). Each process is represented in a window, and can be manipulated independently from the others.

The global structure of the process — all dashed lines for parallel compositions and boxes for unary operators — is attached to the window. It defines sub-windows in which automata are shown. Each automaton is drawn on its own virtual sheet of paper, and no additional virtual sheets are allocated to composed processes. Hence, all virtual sheets have the same size, and a given object (box, arrow) has a unique graphical description (absolute coordinates). It is not modified if the automaton to which it belongs is used as the operand of a complex process. The graphical interface allows the user to move each sub-window of a composed process on the virtual sheet of the automaton it shows, independently from the others. The same is true for zooming. For big descriptions, this mechanism provides an easy way of disposing objects on the screen so that a particular part is well visible, and other parts are hidden or reduced.

The debugger works with the compiled form of a program. It is *symbolic*. The compiler produces a LTS (Labeled Transition System) which contain suf-

[1] X Window System is a trademark of the Massachusetts Institute of Technology.

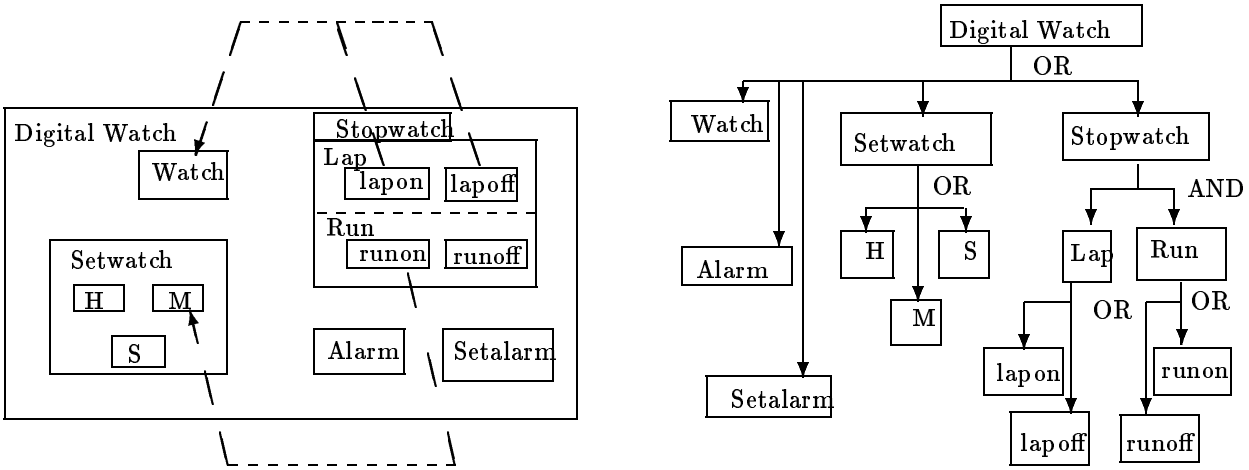

Figure 4: a Statechart and the corresponding state space decomposition

ficient information so that states and transitions of the LTS can be related to sets of boxes and sets of arrows in the program source. The debugger takes a sequence of inputs, either interactively, or in a file, and illustrates the corresponding behaviour of the reactive system by enhancing current states and current transitions. Output events can be stored in a file, or shown on the screen.

Argonaute is connected to graphical tools which work with the compiled form of a program. For instance an interactive graph editor which can be used to produce graphical pictures of the compiled form of a program, when it is reasonably small. Another connected tool allows the description of the system interface and its connection with the compiled reactive kernel, for simulation purpose.

Translators are provided to build postscript or LaTeX pictures from an argos program. If the program has been edited with the graphical editor, it contains graphical information which is simply translated into postscript or LaTeX. If it has been edited textually, it does not contain graphical information, and the translators or the Argonaute editor perform automatic placement of boxes and transitions. Of course the output is not satisfying, but the Argonaute editor or the Idraw tool can be used to rearrange it it interactively. Idraw is a graphical interactive editor in the style of MacDraw, using postscript files.

5 Conclusion

The graphical language Argos for the description of reactive systems is based upon the description of small systems by automata, which are represented graphically. It provides the designer with high level constructs that allow the semantical decomposition of the reactive behaviour into smaller ones. These constructs are given a graphical syntax such that big descriptions can be cut into meaningful parts. This semantical structure is used also in the graphical interactive environment, which functionalities are derived from the possible structures of a program. For instance, parallel components in a system can be manipulated separately in the same window, the zoom can be applied to one component independently from the others, etc.

Argos and Argonaute illustrate the way a language can be designed by taking into account the graphical constraints at the early stages of development. The graphical syntax often guides the choice of the operators (non-parameterized parallel operator, small number of different constructs).

The Argonaute prototype has been used for non-trivial systems (the complete digital watch, parts of communication protocols, real time controllers, the man/machine interface of the Argos editor, ...) and the main conclusions are the following:

- the use of the graphical and the textual syntaxes together is often convenient
- Argos is a very simple language, and some features could be introduced as macro-notations without making the graphical syntax too complex

- the present release of the editor does not allow the insertion of comments, for instance in boxes which represent states. It would be very useful, however. Comments could be attached to states and edited in temporary windows, for instance.

- the complete representation of a big system is seldom necessary in the graphical environment

- the main problem is the absence of a notation for interfaces between components. This information is present in a description, but is not very readable, because it has to be infered from the local event declarations.

One suggestion could constitute a solution for all these problems.

The representation of the whole system with one picture, even if it can be splitted into small parts, and drawn on separate paper sheets, is analogous to the text of a program where procedure calls are replaced by the code of the procedure. It implies a possible duplication of code, but also: no renaming of "parameters" (input and output events for Argos), and bad readability, because the parameter list is replaced by assignments to variables.

We feel that the notion of procedure can be applied in the Argos context. A sub-system is defined completely by its *interface* which gives its input and output events, and by a description of its behaviour. Then a part of a representation can be replaced by the name of a sub-system, followed by the list of effective "parameters", that is the list of input and output events to be used in the system behaviour. This allows the renaming of events, the reusability of a subsystem in different contexts (the same behaviour with different input or output events), a clear notation for sub-system interfaces,...

Moreover, a perspective of research concerns the use of Argos as a "coordinating" language in a multi-language environment for reactive systems. Indeed, Lustre and Esterel are compiled into automata, and Argos deals with automata directly. An automaton in an Argos description may be given in the Argos syntax, but it could also be the output of the Lustre or Esterel compiler. So a compiled program can be used as an automaton operand in Argos. This would be easier with the procedure notation: a procedure has an interface, and its body can be described in Argos or in another language. If it is described in another language, the compiled form of the description must be used.

Investigations about the advantages of graphism in such a multi-language environment have to be done, following the conclusions obtained for Argos. A environment in which text and graphism may be mixed for the description of complex systems seems to be a promising idea, not only because other languages are textual, but also because some parts of a system can be described with an Argos automaton which structure is very simple, whereas its labels are complex, for instance. A textual description of such an automaton is often better than the graphical one.

References

[1] G. Berry, G. Gonthier, *The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation*, ENSMP-INRIA, Sophia-Antipolis, 06565 Valbonne - France (1988).

[2] P. Caspi, D. Pilaud, N. halbwachs, J.A. Plaice, *Lustre, a declarative language for programming synchronous systems*, 14th ACM Symposium on Principles of Programming Languages, Munich, Janvier 1987.

[3] J.C. Fernandez, *An Implementation of an Efficient Algorithm for Bisimulation Equivalence*, Science of Computer Programming, vol. 13, 2-3, may 1990.

[4] D. Harel, *StateCharts : A visual Approach to Complex Systems*, Science of Computer Programming, Vol. 8-3, pp. 231-275 (1987).

[5] D. Harel, *On Visual Formalisms*, CACM vol. 31, no 5 (1988).

[6] D. Harel, A. Pnueli, *On the Development of Reactive Systems*, Logic and Models of Concurrent Systems, Proc. NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, NATO ASI Series F, vol. 13, Springer-Verlag (1985).

[7] F. Maraninchi, *Argonaute: Graphical Description, Semantics and Verification of Reactive Systems by Using a Process Algebra*, Workshop on Automatic Verification methods for Finite State Systems, Grenoble 12-14 June 1989, Springer-Verlag, LNCS 407.

[8] F. Maraninchi, *Argos: a graphical synchronous language for the description of reactive systems*, submitted to Science of Computer Programming.