About the Design of Distributed Control Systems: The Quasi-Synchronous Approach ✧

Paul Caspi[1], Christine Mazuet[2], and Natacha Reynaud Paligot[2]

[1] VERIMAG♣, 2 rue de Vignate, F-38610 Gières
caspi@imag.fr
[2] Schneider Electric, Usine M3, F-38050 Grenoble Cedex 9
{christine_mazuet, natacha_reynaud-paligot}@mail.schneider.fr

Abstract. The European project Crisys✦ aims at improving and formalizing the actual methods, techniques and tools used in the industries concerned with process control, in order to support a global system approach when developing Distributed Control System. This paper focuses on the main result of the Crisys project: the quasi-synchronous approach which is based on the synchronous language Lustre-Scade. The quasi-synchronous methodology provides (1) a complete framework consistent with usual engineering practices for both programming, simulating, testing a distributed system and (2) a robustness properties checker so as to ensure the behavior preservation during the distributed implementation. Both elements are based on a solid theoretical basis.

1 Introduction

Developing Distributed Control System is a major industrial concern since those systems are more and more complex and involved in many safety critical application field. The distribution feature of these systems is not without consequences on both the development process and the exploitation of the system: the global behavior of the system is more complex since distribution introduces new operating modes — abnormal modes, when a computing site is down for instance— and questions about the synchronization of the different computing sites. Distributed Control Systems (DCS) are hard to design, debug, test and formally verify. These difficulties are closely related to a lack of global vision at design time. Moreover, the implementation would be eased using automatic methods of distribution which guarantee that the behavior of the whole system is preserved.

To face up to these difficulties engineers have developed solutions of their own. Their solutions are essentially pragmatic and based on engineering rules. But a theoretical basis is lacking if we want formally to understand, design and verify Distributed Control Systems when applied to critical fields.

The European project Crisys originates from this industrial need. The overall goal of the Crisys project is to improve, unify and formalize the actual methods, techniques

✧ This work has been partially supported by Esprit Project CRISYS (EP 25514).

♣ VERIMAG is a joint laboratory of Université Joseph Fourier (Grenoble 1), CNRS and INPG.

✦ http://borneo.gmd.de/~ap/crisys/

U. Voges (Ed.): SAFECOMP 2001, LNCS 2187, pp. 215–226, 2001.

and tools used in the industries concerned with the process control, in order to support a global system approach when developing Distributed Control System. This paper focuses on the main result of the Crisys project, the quasi-synchronous approach which is based on the synchronous language Lustre [1] and the associated tool Scade [2]. This approach is dedicated to a special class of DCS: in the Control field, most of DCS are organized as several periodic processes, with nearly the same period, but without common clock, and which communicate by means of shared memory through serial links or field busses. This class of DCS is quite clearly an important one in the field and thus deserves special attention.

The paper is organized as follows: section 2 presents an overview of the Crisys methodology based on the quasi-synchronous approach. Then, section 3 briefly describe the Lustre-Scade tool-set for designing distributed systems. In section 4, we focus on the robustness properties which guarantee that the centralized behavior of the system is preserved when distributing the system according to the chosen architecture. Section 4 describes the application of the Crisys methodology to an industrial case study. Finally, section 6 concludes with future work.

2 Overview

2.1 Industrial Practices

The Lustre-Scade language is largely and successfully applied to the development of distributed control systems [3] [4] [5]. But so far, the engineers make use of Lustre-Scade to design single components of a DCS. Schematically, the industrial software development proceeds as follows (Fig. 1):

- The specification phase involves both the functional description —i.e. the behavior of the whole system independently of its architecture— and the distribution protocol which specifies the physical implantation of the functional components. So far, the solution to design robust distributed systems —i.e. whose functional behavior is preserved when distributing it— are pragmatic and based on the engineers know-how.
- Each component is developed separately with Lustre-Scade. The global view of the system is no longer preserved. Moreover, there is usually a breaking in the tool chain between this step and the previous one.
- Finally, pieces of code resulting from the previous step are plug into the physical target and connected by means of network (e.g. [6]).

The goal of the Crisys project is to improve this development process based on Lustre-Scade, by formalizing the industrial practices and providing support of tools.

2.2 The CRISYS Methodology

The methodology defined within the Crisys project is shown on Fig. 2.

1. From the functional specification, a Lustre-Scade model of the global system is developed. At this stage, this functional model can be simulated, formally verified and tested.

2. The second step consists in completing the functional model with the distribution protocol. Then, the resulting architecture is checked by means of the *robustness properties analyzer*. This tool aims at guaranteeing that the behavior of the distributed system is consistent with the behavior of the centralized one. The analysis is based on three robustness properties: stability, order-insensitivity, and confluence (§4).
3. A distribution scheme being acceptable, it is possible to test, simulate, formally verify the distributed system in a realistic way by means of the *environment emulation library*. It is important to note that the same tools applied to the centralized model and to the distributed one allowing the comparison of their behavior.
4. Finally, the code corresponding to each component is generated together with some communication elements provided by the *communication library for target*.

An application of this methodology is presented in section 5.

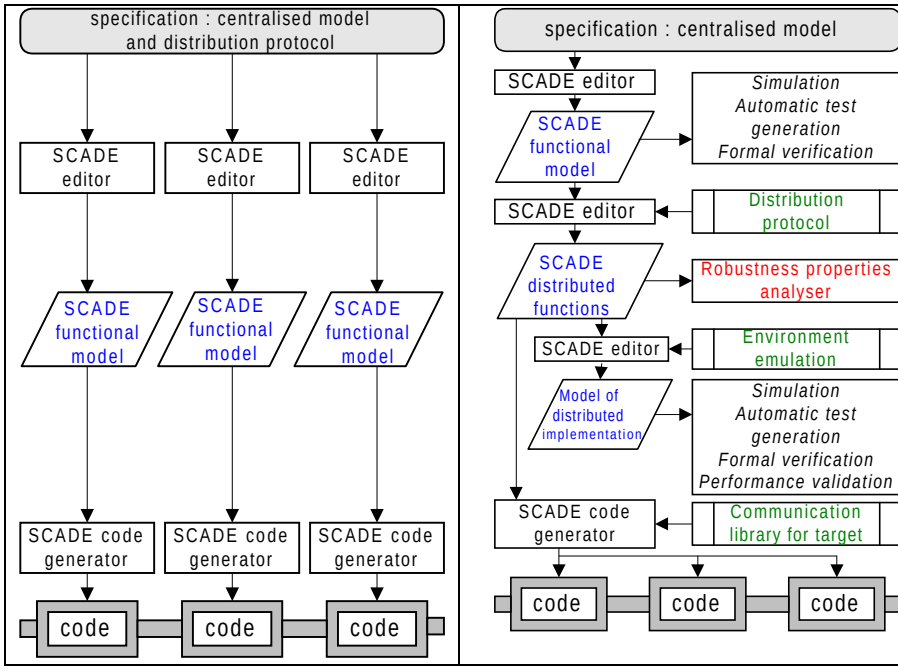

Fig. 1. The industrial software development

Fig. 2. The CRISYS methodology

The valuable consequences of the Crisys methodology are:

- The global view of the distributed system is preserved as long as possible during design. It can be simulated and tested as a whole.
- The robustness properties analyzer based on theoretical foundations formalizes the pragmatic and intuitive solutions achieved by engineers to design robust DCS.

- The same framework can be used for programming, simulating, testing and proving properties of a distributed system. This result makes the comparison between the behavior of the centralized and the distributed system possible.

The Crisys work has first focused on the use of Lustre-Scade for designing a DCS as a whole, i.e. for describing the Scade distributed model (§3.3). Then, the second step has concentrated on the robustness properties analysis (§4).

3 Background

3.1 The Lustre Language and the Scade Tool

Lustre [1] is a synchronous data-flow language. Each expression or variable denotes a flow, i.e., a function of discrete time. The Lustre equation x=2*y+z means: "*at each instant t,* x(t)=2*y(t)+z(t)". Lustre provides a special operator "previous" to express delays: "y=pre(x)" means that at each time t≠0 we have y(t)=x(t-1), while the value of y at time 0 is undefined. To initialize variables, Lustre provides the "followed by" operator: "z=x→y" means that z(0)=x(0) and z(t)=y(t) for each time t≠0.

A Lustre program is structured into *nodes*. A node contains a set of equations and can be elsewhere used in expressions. It may be that slow and fast processes coexist in a given application. A sampling (or filtering operator) *when* allows fast processes to communicate with slower ones. Conversely, a holding mechanism, *current* allows slow processes to communicate with faster ones.

Scade[1] (formerly SAGA [2]) is a software development environment based on Lustre, which provides a graphic editor. Its main features are the top-down design method, the data-flow network interpretation, and the notion of *activation condition*.

An example of Scade diagram is given on Fig. 3. *CONTROL* is a cyclic program which reads sensors and controls actuators. Its inputs and outputs are sampled according to the boolean condition *clock*: intuitively, if *clock* is *true* then *CONTROL* computes its outputs, else the outputs are maintained to their previous values. Default values are required in case *clock* is *false* at the very first cycle.

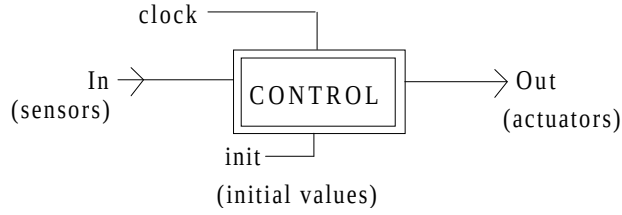

Fig. 3. Example of Scade diagram

The Scade environment includes an automatic C code generator and a simulator. It is also connected to several tools (§3.2).

[1] Scade is commercialised by the Telelogic company.

3.2 The Lustre-Scade Tool-Set

Several tools have been developed to improve and facilitate the design and the verification of Lustre-Scade programs. For example, Lesar [7] and Lucifer[2] [8] for formal verification, Matou for describing Mode-Automata [9], Lurette [10] for automatic test cases generation. Scade can be also connected to ISG[2] [11] for performance validation.

Let us concentrate on the automatic generation of test sequences with Lurette. The automatic generation of test cases follows a black box approach, since the program is not supposed to be fully known. It focuses on two points [10]: (1) generating relevant inputs, with respect to some knowledge about the environment in which the system is intended to run; (2) checking the correctness of the test results, according to the expected behavior of the system. The Lustre synchronous observers[3] describing assumptions on the environment are used to produce realistic inputs; synchronous observers describing the properties that the system should satisfy (§2.3.1) are used as an oracle, i.e. to check the correctness of the test results. Then, the method consists in randomly generating inputs satisfying the assumptions on the environment [10].

The Lurette tool takes as input both observers —one describing the assumptions and one describing the properties— written in Lustre-Scade, and two parameters: the number of test sequences and the maximum length of the sequences. An experimentation of Lurette is presented in section 5.

3.3 The Quasi-Synchronous Approach

The above language and tools accurately match the needs of single cyclic components. But how can they be used to design a distributed system as a whole? The first step of the Crisys work aimed at formalizing the description of a DCS by means of the Lustre-Scade language [13].

First let us remind ourselves the main features of the quasi-synchronous class of DCS: process behave periodically, they all have nearly the same period but no common clock and they communicate by means of shared memory. These features can be formalized by means of the Lustre-Scade primitives (Fig. 4):

- Each processes has got its own clock represented by an activation condition. For example, on Figure3, process *S1* is activated each time its clock *c1* is true.
- Shared memories are modelled through both the activation condition and delays (pre, ->).

Finally hypothesis on clocks can be implemented through a Lustre-Scade program: the quasi-synchronous program generates clocks with nearly the same period (§5.3.3, Fig. 12) This is one of the component of the environment emulation library (Fig. 2).

[2] Partly developed within the framework of the Crisys project.
[3] Synchronous observers are acceptors of sequences [12].

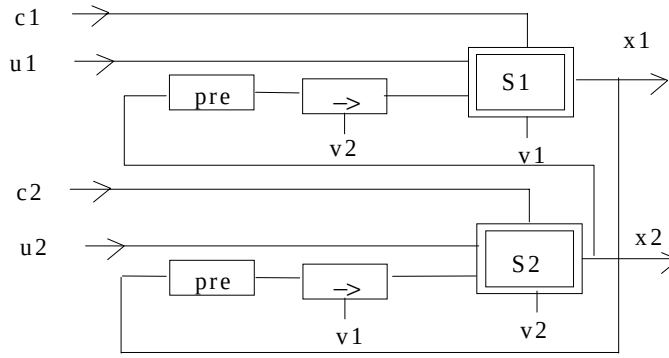

Fig. 4. A distributed system block diagram

4 Towards a Robust Distribution

Given the Lustre-Scade model of the distributed system, which additional checks have to be performed so as to ensure the behavior preservation during the implementation? Three constraints —called robustness properties— have been identified in order to guarantee that the behavior of the distributed system is the same as the centralized one. These checks are implemented through a tool —the robustness properties analyzer— which is one of the key element of the Crisys methodology.
In this chapter, we present in an informal way the three robustness properties. The theoretical details can be found in [14] [15] [16].

4.1 Stability

It is likely indeed that distributed programs will have to run faster in order to produce behaviors comparable to those of centralized programs. But running a synchronous program faster on the same inputs will in general deeply modify its behavior. This is why we may expect it easier to distribute stable systems rather than unstable ones, stable systems being those that can run faster without too much changing their behaviors.

In other words, a stable system will stabilize when the inputs do not change. Figure 5 gives an example of non-stable system: when *u* remains true, the output *x* is indefinitely oscillating between true and false. Let us now suppose a redundant system involving two sub-systems defined by the equation of Figure 3. The oscillation made the comparison of both results meaningless.

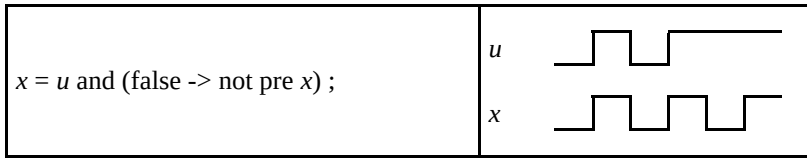

Fig. 5 . Example of a non stable system

4.2 Order Insensitivity

Another feature of distributed systems is that their components are not computed in a parallel synchronous fashion but in a sequential (chaotic ordered) way. A system is order-insensitive if its behavior does not depend on the order computations are performed. Figure 6 gives an example of an order sensitive system. As regards the centralized behavior (Fig 6.a), the output *y* reaches the true value because the input *u* is true and the previous value of *x* is false. Let us now assume that computations of *x* and *y* are performed on two different processors running with different time cycles. If the *x* value is computed and sent to the other processor before *y* is computed (Fig. 6.b), then *y* can no more reach the true value because its calculation refers to the latest value of *x* which is now true.

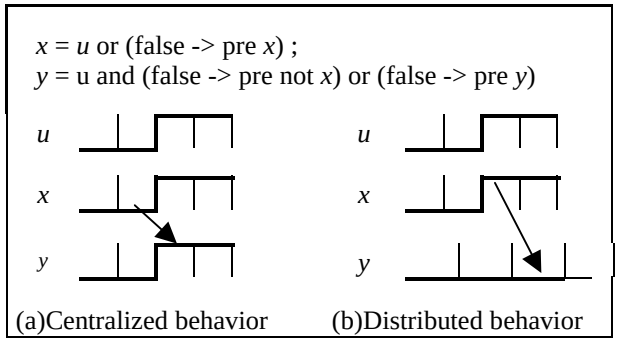

Fig. 6 . Example of an order sensitive system

A stronger property called *state decoupling* [14] is satisfied when each component depends only on its internal state.

4.3 Confluence

Another desirable property for distribution is *confluence*. It means that input changes can be arbitrarily composed while yielding the same final state. The order the inputs are read does not have to imply different behaviors. An example of a non confluent system is given on Figure 7. The outputs *x* and *y* are obviously equal (when they are computed in a centralized manner). But if the inputs *u* and *v* are sampled according to the dotted line then *x* and *y* differ from each other. The centralized behavior is no longer preserved.

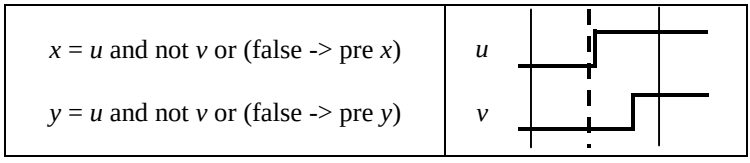

Fig. 7 . Example of a non confluent system

However, confluence is a very restricted property and we cannot limit ourselves to distributing confluent functions. We may need to strengthen this definition by considering *local confluence* [14].

5 Case Study

The Crisys methodology (§2.2) is now illustrated on a real case study from Schneider. Through this experimentation, our aim is to check the feasibility and the benefits of the quasi-synchronous approach based on Lustre-Scade.

5.1 Introduction

The Water Level Control System (WLCS) is a system controlling the water level in a steam generator. This system is aimed to be implemented in power plants (nuclear or thermal). Basically, the WLCS operates on two valves so that the water level is unchanged. Several sensors are present all along the steam generator to measure the water level, the flow, the temperature and the thermal power.

The WLCS is a typical loop system. It is made of three steps (Fig. 8):

- the water level control loop that provides a water flow set point,
- the water flow control loop that provides the valves position set point,
- the valves position control loop that controls the valves.

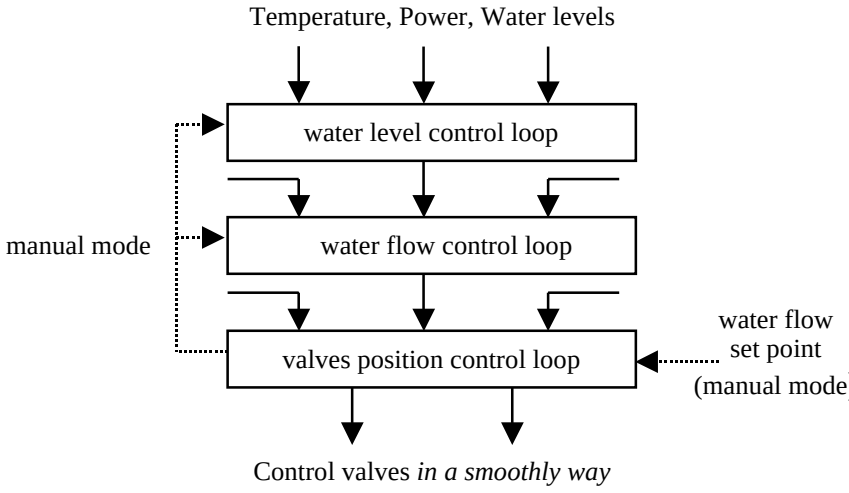

Fig. 8. WLCS functional view

One of the main requirement is that the valves have to be controlled in a smoothly way in order to avoid discrepancies. During the experimentation, a particular attention has been taken on the switching between the automatic and the manual mode, since this change may imply discrepancies.

The WLCS has been developed with SCADE using the CRISYS methodology (Fig. 2). The experimentation has been conducted through different steps:

- At first, the WLCS has been designed and simulated in the centralized way.
- Then a distributed architecture has been proposed and analyzed.
- Finally, the distributed system has been simulated and its behavior has been compared to the centralized one.

5.2 The Centralized System: Design and Simulation

The centralized system has been designed with the SCADE tool according to the functional view showed on Figure 8. Moreover, in order to simulate the system as if it was physically implemented, the behavior of the different sensors has been designed, i.e. the system is simulated in closed loop (Fig. 9).

The automatic generation of test sequences, Lurette, has been used to simulate the system.

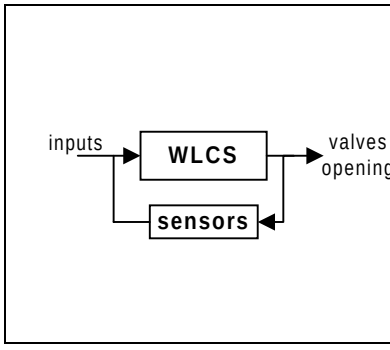

Fig. 9. The closed loop system

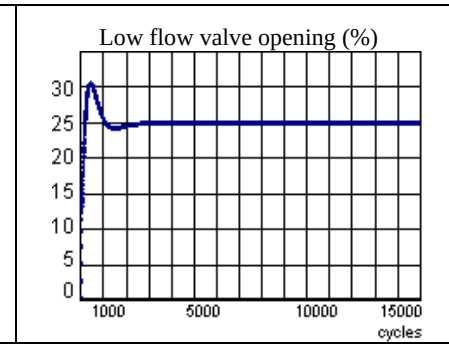

Fig. 10. Example of result

An example of results is given on Figure 10. We can see that after the initialization phase, the valve opening stabilizes at 25 % after 2000 cycles (i.e. 500 seconds).

5.3 The Distributed System

Architecture analysis. The architecture of the system has been defined by the client for performance reasons. The system is made of two sub-systems which communicate with each other (Fig. 8):

- the first sub-system involves the water level control loop and the water flow control loop,
- the second sub-system involves the valves position control loop.

In order to guarantee that the behavior of the distributed system is preserved, this architecture has been analyzed with the *robustness properties analyzer* (see §4.2). The result of the tool is that the WLCS is stable, order-insensitive and confluent as far as the proposed architecture is concerned.

Scade design. According to the quasi-synchronous approach (§3.3), each sub-system has got its own clock representing its own cycle. The WLCS is composed of two sub-systems which have different clocks (Fig. 11):

- the water level control loop and the water flow control loop have the same clock (CLK1),
- the water position control loop has a different clock (CLK2).

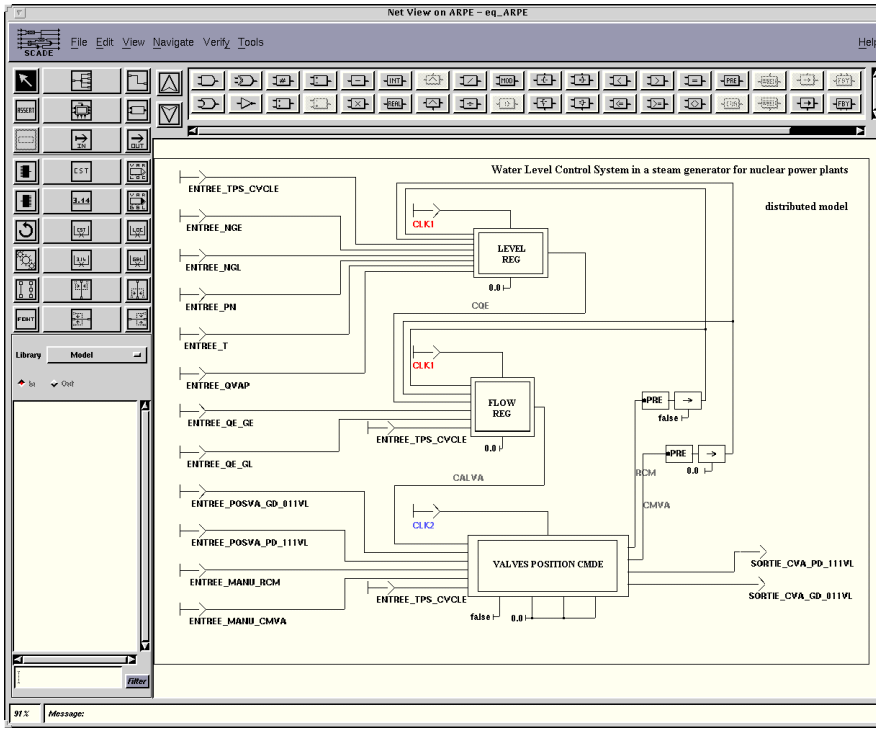

Fig. 11. SCADE distributed model

Simulation. The goal of the simulation is to check the behavior of the distributed system in a realistic way. Clocks are generated according to the quasi-synchronous hypothesis (i.e. periodic real time clocks of each process are subject to drifts) by means of the *environment emulation library*. An example of the clocks used for the two WLCS's sub-systems is given on Figure 12. These clocks are pessimistic since data can be lost.

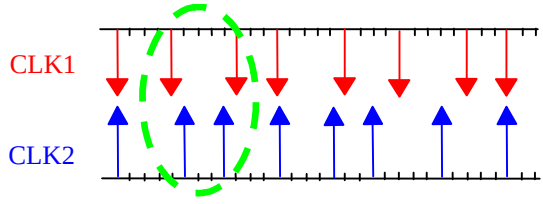

Fig. 12. Quasi-synchronous clocks

5.4 Results and Comparisons

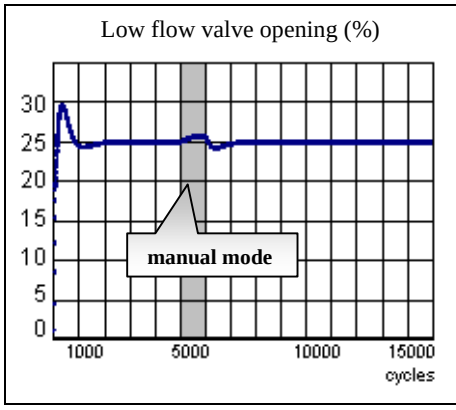

Fig. 13. Centralized system

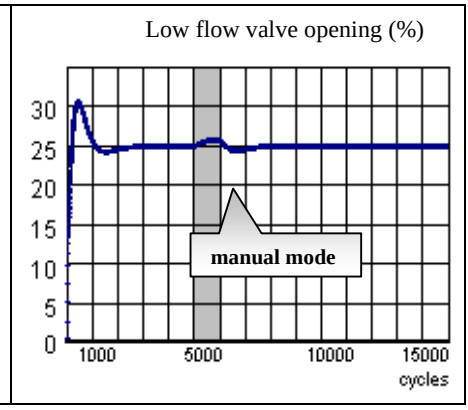

Fig. 14. Distributed system

The behaviors of the centralized system (Fig. 13) and the distributed system (Fig. 14) are similar: first, the low flow valve opens up to 30% and then stabilizes around 25%. In manual mode, the operator increases the opening set point. When coming back to the automatic mode, the valve opening oscillates and then stabilizes again at 25%. The control is performed in a smoothly way as required for the centralized and the distributed systems.

As regards the distributed case, we can note that the time response is slower due to the communication delays between the two sub-systems during the simulation.

6 Conclusion and Future Work

The experimentation shows the feasibility and the benefit of the quasi-synchronous methodology. An additional experimentation on a case study from the aircraft industry enforces this conclusion. The quasi-synchronous methodology provides:

- a global view of the Distributed Control System which can be designed and simulated within the same environment, in consistency with the usual engineering practices;
- an automatic robustness analyser which aims at guaranteeing that the behaviour will be preserved when distributing the system according to the target architecture.

These two points are key elements to reduce the industrial development costs.

The next steps of the work are twofold:

- some tools need to be improved so that they can easily be integrated in the industrial development process;
- the experimentation on the Schneider case study will be continued untill the final implementation of the generated code.

Acknowledgement

The authors would like to thank all the partners involved in the Crisys project.

References

1. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305-1320, September 1991.
2. J.L. Bergerand and E. Pilaud. SAGA: a software development environment for dependability in automatic control. In *Safecomp'88*. Pergamon Press, 1988.
3. A. Billet and C. Esmenjaud. Software qualification: the experience of french manufacturers. *International Conference on Control and Instrumentation in Nuclear Installations*, INEC Cambridge, Great Britain, April 1995.
4. J.-M. Palaric and A. Boué. Advanced safety I&C system for nuclear power plants. In *ENC'98 World Nuclear Congress*, Nice, France, October 1997.
5. D. Brière, D. Ribot, D. Pilaud, and J.L. Camus. Methods and specification tools for Airbus on-board systems. In *Avionics Conference and Exhibition*, London, December, 1994. ERA Technology.
6. A. Boué and G. Clerc. Nervia: a local network for safety. In *IAEA Specialist Meeting on Communication and data transfer in Nuclear Power Plants (CEA/EDF/FRAMATOME editors)*, Lyon, France, April 1990.
7. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous dataflow language Lustre. *IEEE Trans. on Software Engineering*, 18(9):785-793, September 1992.
8. M. Ljung. Formal modelling and automatic verification of Lustre programs using NP-Tools. In *Crisys deliverable n°CMA/ 999909*, October 1999.
9. F. Maraninchi and Y. Rémond and Y. Raoul. MATOU : An Implementation of Mode-Automata into DC. In *Proceedings of Compiler Construction*, Berlin, Germany, 2000
10 P. Raymond, X. Nicollin, N. Halbwachs and D. Weber. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.
11 R.Gerlich. An Implementation and Verification Technique for Distributed systems. In: *F.Cassez, C.Jard, B.Rozoy, M.Ryan (eds.), Proceedings of the Summer School "Modelling and Verification of Parallel Processes (MOVEP'2k)*, Ecole Centrale de Nantes, June 2000, p. 285 - 296.
12. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993.
13. Paul Caspi, Christine Mazuet, Rym Salem, and Daniel Weber. Formal Design of Distributed Control System with Lustre. In *proceedings of the 18th International Conference on Computer Safety, Reliability and Security (SAFECOMP'99)*, Toulouse, France, September 27-29, 1999.
14. P. Caspi. The quasi-synchronous approach to Distributed Control Systems. *Crisys deliverable n° CMA/ 009931*, May 2000.
15. M. Yeddes and H. Alla. Checking Order-insensitivity using Ternary Simulation in Synchronous Programs. *IEEE, ISPASS 2000*, Austin (USA), pp.52-58, 24-25 April 2000.
16. M. Yeddes, H.Ala, and R. David. On the Supervisory Synthesis for Distributed Control of Discrete Event Dynamic Systems with Communication Delays. In *Proceedings of the 1999 IEEE ISIC, Massachusetts (USA)*, pp. 1-6, 1999.