Formal Design of Distributed Control Systems with Lustre ✧

Paul Caspi[1], Christine Mazuet[2], Rym Salem[1] and Daniel Weber[2]

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

Abstract. During the last decade, the synchronous approach has proved to meet industrial needs concerning the development of Distributed Control Systems (DCS): as an example, Schneider Electric has adopted the synchronous language Lustre and the associated tool Scade for developing monitoring systems for nuclear power plants. But so far, engineers make use of Lustre-Scade for designing separately single components of a DCS. This paper focuses on the use of Lustre-Scade for designing DCS as a whole. Two valuable consequences of this approach are that (1) the same framework can be used for both programming, simulating, testing and proving properties of a distributed system, and (2) the proposed approach is fully consistent with the usual engineering abstractions concerning smooth signals.

1 Introduction

Control systems are of growing importance as they are involved in many safety critical industrial applications: civil aircraft, ground transportation, nuclear power plant, etc. In this field, a lot of activity has been devoted to ensuring and improving hardware and software reliability. Concerning software development, fault avoidance has always been, beside fault tolerance, an important issue: constrained design process, intensive simulation and testing and even formal methods (e.g. [1], [4]) have proved to be good candidates to answer this problem.

Another feature of control systems is that they are often distributed for quite obvious reasons of performance, fault-tolerance, and sensor/actuator location. Distribution 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. Those difficulties are closely related to a lack of global vision of a system when designing it.

✧ 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.

However, the distributed systems which are found in the Control field are quite different from those that are addressed in other fields of Computer Science, like protocols, operating systems, data bases, etc. Most of them 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 (e.g. [7]). This class of DCS is quite clearly an important one in the field and thus deserves special attention. This is why we propose to give it a special name, for instance *"Communicating Periodic Synchronous Processes"* or *CPSP* for short.

During the last decade, the synchronous approach [4] has been largely and successfully applied to the development of such distributed control systems. Based on clean mathematical principles, one of its salient benefit is its ability to support formal design and formal verification methods. As regards the synchronous data flow language Lustre [9] and the associated tool Scade [5], several real world systems have been achieved among which monitoring systems for nuclear power plants designed by Schneider Electric [6] [12], the "fly-by-wire" system of Airbus aircrafts [8] and the interlocking system of the Honk Kong subway designed by CS Transport. So far, engineers make use of Lustre-Scade for designing separately single components of a DCS. As regards designing distributed systems as a whole, they have developed pragmatic solutions based on engineering recipes.

In this paper, we intend to show that this Lustre-Scade approach is not restricted to apply only on single components; it can be extended to globally handle this CPSP class of DCS, thanks to the Lustre sampling, holding and assertions mechanisms. One valuable consequence of this result is that the same framework can be used for both programming, simulating, testing and proving properties of a distributed system, thus limiting the risk of errors due to an unformalized design process.

The paper is organized as follows: section 2 briefly describes the Lustre language, the Scade tool and related works on verification. Based on this background, section 3 shows how to represent the CPSP class of DCS within this framework. Section 4 focuses on a case study. Finally, section 5 concludes with future work.

2 Background

2.1 The Lustre language [9]

2.1.1 Basic concepts

A Lustre program has a cyclic behaviour, and that cycle defines a sequence of times. Any variable or expression denotes a flow, i.e. a possibly infinite sequence of values related to the cyclic behaviour of the program.

Usual operators operate pointwise over flows; for instance 1 + 2 is the flow [3, 3, 3, ...]. Similarly,

c	t	f	t	...
x	x_0	x_1	x_2	...
y	y_0	y_1	y_2	...
if c then x else y	x_0	y_1	x_2	...

A unit delay operator *fby* "followed by" (actually -> *pre* in Lustre) can be represented by the diagram:

x	x_0	x_1	x_2	...
y	y_0	y_1	y_2	...
x fby y	x_0	y_0	y_1	...

2.1.2 Advanced concepts

A program is structured into *nodes*. A node contains a set of equations and can be elsewhere used in expressions; for instance:

```
node integr(x:int) returns (y:int)
let
y = (0 fby y) + x;
tel
...
z = integr (x fby z) + u;
```

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:

c	f	t	f	t	...
x	x_0	x_1	x_2	x_3	...
x when c		x_1		x_3	...

Conversely, a holding mechanism, *current* allows slow processes to communicate with faster ones:

c	f	t	f	t	...
x	x_0	x_1	x_2	x_3	...
y		y_0		y_1	...
current(c, x) y	x_0	y_0	y_0	y_1	...

As we can see in the diagrams above, *when* discards its input *x* when the input condition *c* is false. Conversely, *current* fills the holes created by *when* with the input value *y* it got the last time the condition *c* was true, if any, and otherwise with an initialising sequence *x*.

Always "true" boolean expressions can be asserted for several purposes, for instance for expressing non independent input properties; for instance

```
assert (c or (true fby c) )
```

says that *c* will not stay "false" for more than one time unit. We shall see in the sequel how this feature can be used to express properties of independent clocks sharing the same period.

2.2 The Scade tool

Scade[1] (formerly SAGA [5]) is a software development environment based on Lustre, which provides a graphic editor. Some aspects are similar to SADT: the top-down

[1] Scade is commercialised by the Verilog company.

design method, the data-flow network interpretation, and the notion of *activation condition*.

An example of Scade diagram is given on Fig. 1. *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[2]. Default values are required in case *clock* is *false* at the very first cycle.

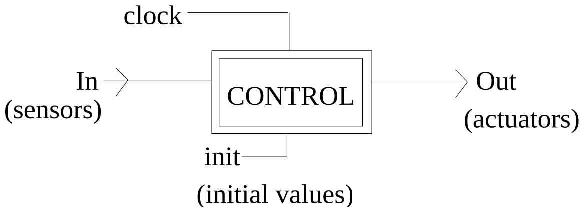


Fig. 1. Example of Scade diagram

The Scade environment includes an automatic C code generator and a simulator. It is also connected to several tools, e.g. Lesar for formal verification of properties (§2.3.1) and Lurette for automatic generation of test cases (§2.3.2).

2.3 Formal verification and automatic testing

As noted in the introduction, control systems often concern critical applications, and thus program verification is a key issue.

2.3.1 Formal verification of properties

Formal verification of properties focuses on safety properties, which states that a given situation should never appear, or that a given statement should always hold. Such properties can be easily expressed with Lustre [3] [10] [11].

Then, the verification principle is based on model checking: the proof is made by an exhaustive exploration of a finite abstraction of the system. However, at that point, an important task is to provide a description of how the environment of the system behaves. Actually, the environment obeys some rules which restrict its possible behaviour. The verification tool would certainly not achieve checking the system without being aware of such an information. Assumptions are therefore necessary: in Lustre, this is done using the assertions mechanisms (§2.1.2).

Properties and assumptions are expressed by means of *synchronous observers* [11]: synchronous observers are programs implementing acceptors of sequences. Then, the program and the observers are gathered in a verification program [10], and the verification tool Lesar decides whether the properties is satisfies; if not, it gives a scenario which leads to violate the property. The main limitation of this approach is the "size explosion" problem. An experimentation is presented in §4.3.

[2] Formally, the equivalent Lustre expression is: if *clock* then current(*CONTROL*(*In* when *clock*)) else (*init* fby *Out*)

2.3.2 Automatic generation of test sequences

Program testing is complementary to formal verification: it aims at finding bugs but can not provide any absolute positive results. 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 [14]: (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 behaviour of the system.

The Lustre synchronous observers describing assumptions on the environment (§2.3.1) 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 [14].

A prototype tool called Lurette has been developed. It 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 is presented in §4.4.

3 Application to distributed systems

The above tools quite accurately match the design needs of cyclic programs. Let us see now how we can use them so as to match the needs of our special CPSP case of distributed systems. Let us recall roughly some of the main features of this CPSP class: processes communicate by means of shared memory, they behave periodically and they all have nearly the same period but no common clock. In this chapter, we progressively formalise theses features by means of the Lustre-Scade language. First, we model a shared memory, and based on this, we formalise distributed systems without any hypothesis on clocks. Then, we focus on the relation between clocks: we express the fact that two processes have nearly the same period in order to fit our CPSP class of systems.

3.1 Shared memory

First, we can see that the Lustre sampling primitive, *when*, can be used to model the " reading in a shared memory" operation. It suffices to look at the corresponding diagram of section 2.1.2: let *x* be the sequence of values held in the memory, and the true values of *c*, the instants at which a location reads this memory. Then, obviously, "*x* when *c*" is the sequence of values read in this location.

Conversely, the "current" operation can be used to model the "writing in the shared memory" operation: looking at the corresponding diagram (§2.1.2), we can use the *c* sequence to represent the instants at which the memory is written, *y* will correspond to the written values and a constant sequence *x* will be used to account for the initial value of the memory. Then, "*current(c,x)y*" will be the current content of the memory. However, this very simple approach is not fully satisfactory because it allows for instantaneous dialogs which do not seem feasible, such as reading and writing at the same time and instantly getting the written value. This is why we insert a delay —by means of the *fby* operator— in the write operation in order to forbid this event:

```
write(v,cw,u)=v fby(if cw then current u else write(v,cw,u))
```

where the delay accounts for short[3] undetermined transmission delays. The behaviour of the shared memory can be represented by the diagram:

cw	t	f	t	f	...
u	u_0		u_1		...
current u	u_0	u_0	u_1	u_1	...
write(v,cw,u)	v	u_0	u_0	u_1	...

3.2 Distributed programs

We are thus able to formalise distributed programs communicating by means of shared memory. Fig. 2 gives the Lustre expression and the corresponding block diagram view for a two location program. An alternative block diagram view is shown at Fig. 3 using the Scade concept of activation condition.

This system corresponds to a possible implementation where each process atomically reads all values issued by other processes. If this is not the case, individual read and write clocks have to be introduced, leading to a more complex, but also tractable program.

However, this formalisation is likely to be inefficient: it will probably not allow properties to be checked, if clocks are considered as free independent boolean sequences. It is here that periodicity is important and has to be taken into account.

```
Const v1:type1; v2:type2;
node S1 (u1:type1, x2:type2)
returns (x1:type1);
node S2 (u2:type2, x1:type1)
returns (x2:type2);
node Sd (u1:type1, u2:type2,
c1, c2:bool)
returns (x1:type1, x2:type2);
let
x1=S1((u1,
      write(v2,c2,x2))when c1);
x2=S2((u2,
      write(v1,c1,x1))when c2);
tel
```

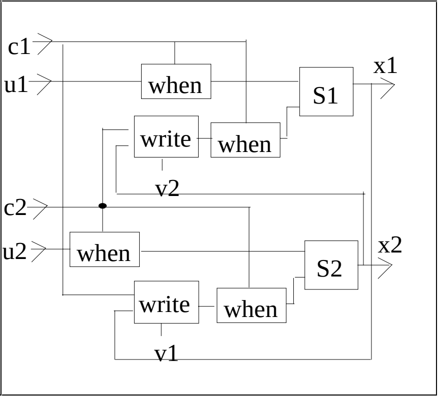


Fig. 2. A distributed system program

[3] Significantly shorter than the periods of read and write clocks. If longer transmission delays are needed, modelling should be more complex.

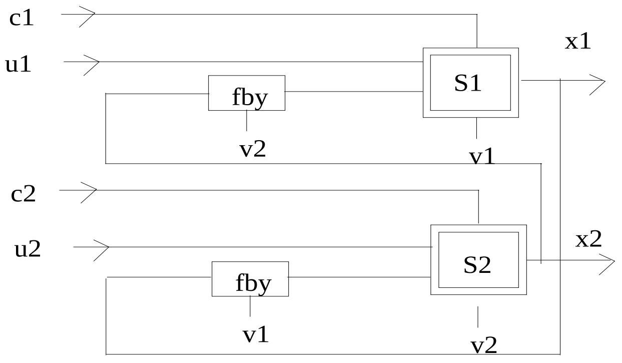


Fig. 3. An alternative equivalent distributed system block diagram

3.3 Formalising periodic clocks

This could be done in some real-time framework, such as timed automata [2] but, for the sake of simplicity, we prefer here to characterize the fact that two independent clocks have approximately the same period by saying that:

Any of the two clocks cannot take the value "t" (true) more than twice between two successive "t" values of the other one.

This can be formalised by saying that the boolean vector stream composed of the two clocks should never contain the subsequence:

$$\begin{bmatrix} t \\ - \end{bmatrix} \bullet \begin{bmatrix} f \\ f \end{bmatrix}^* \bullet \begin{bmatrix} t \\ f \end{bmatrix} \bullet \begin{bmatrix} f \\ f \end{bmatrix}^* \bullet \begin{bmatrix} t \\ - \end{bmatrix}$$

nor the one obtained by exchanging coordinates. (Here, — is a wild card representing any of the two values {t,f}.

Now, such regular expressions yield finite state recognizability and can be associated a finite-state recognizing dynamic system *Same_Period* [4]. Fig. 4 gives an example of clocks satisfying the *Same_Period* property: it may happen that two ticks of clock1 (resp. clock2) are inserted between two ticks of clock2 (resp. clock1). As a matter of fact, data exchanged between the corresponding processes can be lost of duplicated. This protocol does not seem reliable since data can be lost: but we'll see in §3.5 that as far as smooth signals are concerned, this protocol fits engineers practices.

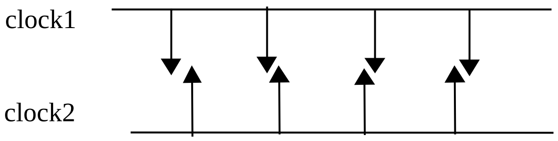


Fig. 4. Periodic clocks

[4] This can be automatically generated in Lustre, thanks to the Reglo tool [13].

3.4 Some consequences of periodicity

Intuitively, if read and write clocks have the same period, the read value should not be "too far" from the written one. This can be formalized as the following theorem:

Theorem on Same_Period

```
                    Same_Period(cw, cr)
implies (           write (v,cr,u') = write (v,cw,u)
                or  write (v,cr,u') = write (v,cw,v fby u)
                or  write (v,cr,u') = write (v,cw, v fby(v fby u))   )
```

where u' = write(v,cw,u) when cr.

This theorem means that any used internal value must have been produced within a two period time interval. It is automatically proved by the Lustre prover Lesar [9].

3.5 The case of smooth signals

A possible use of this theorem is for smooth (sampled) signals. Given a physical phenomenon and knowing an upper bound on the absolute value of its derivative allows finding a period such that:

$$|u - v \text{ fby } u| < \varepsilon$$

This allows us to state the "sampling" property:

$$|\text{ write } (v,cr,u') - \text{ write } (v,cw,u) \,| < 2\varepsilon$$

This kind of sampling property may help in proving distributed sampled system properties. An example of such property can be found in [3]. We believe this is a kind of abstraction, engineers are used to and which explains why such a CPSP architecture is popular among them.

4 Case study

The method described above is now illustrated on a case study. Through this experimentation, our aim is to study the applicability of the tools based on Lustre — namely Scade, Lesar, Lurette— when applied to distributed systems. First, we apply the proposed formalization: we'll see that the *activation condition* and the *assertion* mechanisms provide a natural way to design distributed systems. Then, we focus on the verification on such a distributed system by applying formal verification and automatic test generation.

4.1 Introduction

The proposed example is that of a single line on which trains can go in opposite directions. The control program has to ensure that no accident happens. We consider the global system composed of three main parts:

- the physical part (Fig. 5): the single line, four tracks, a sensor of train presence on each in_track, switches, sensors for switch presence, actuators controlling switches and traffic lights, and the trains.
- the control part: a Lustre-Scade distributed program which reads the sensors and controls actuators (traffic lights and switches).
- the train drivers who control the train movements. They are supposed to obey traffic lights and not to move the train backward.

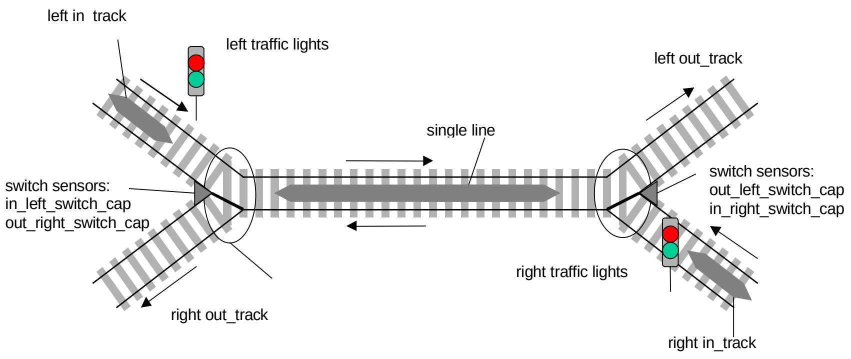


Fig. 5. The single line

4.2 The distributed control system

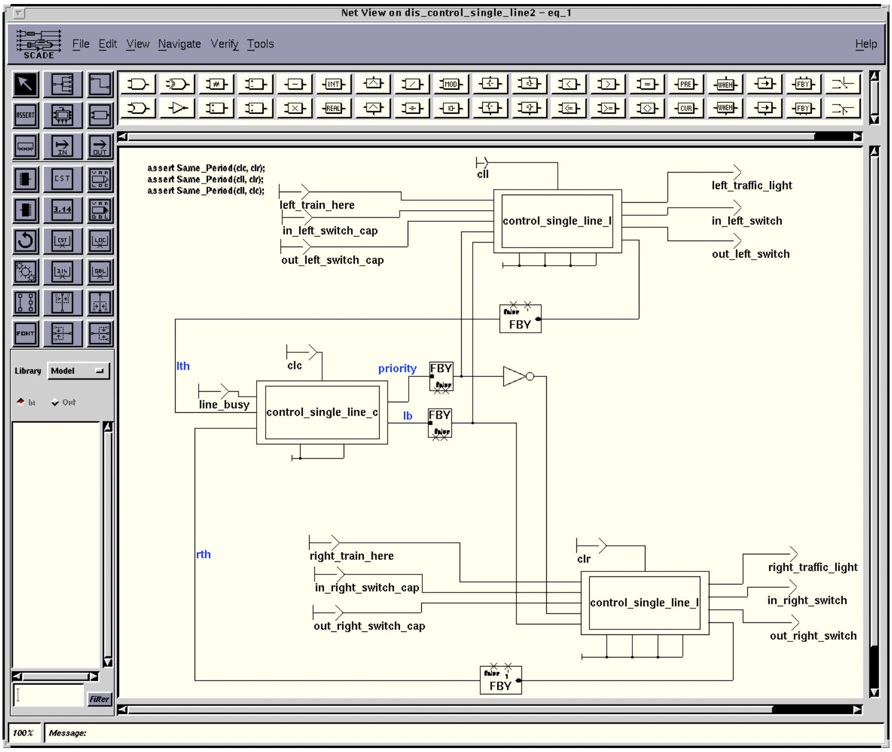
Given the presence sensors, the control decides the switches position and the state of traffic lights. When trains are present on their in_tracks, a selection of the train to pass must be done with respect to some priority strategy. Such a strategy must ensure that there is no deadlock nor starvation. Of course, if only one train is present it will be selected to pass.

The hardware architecture of the system is composed of three units respectively located on the left in_track, on the right in_track and on the single line. They communicate through networks by means of shared memory. Sensors located on the left track (resp. right track) are connected to the left unit (resp. right unit). The sensor located on the single line is connected to the third unit.

The software architecture follows the hardware one:

- management of the left direction (resp. right direction) is implemented in the left (resp. right) unit,
- the function managing the priorities between the two direction is implemented in the central unit.

The software is developed with Lustre-Scade. The top level diagram is shown at Fig. 6.



```
assert Same_Period(clc, clr);
assert Same_Period(cll, clr);
assert Same_Period(cll, clc);
```

Fig. 6. The distributed control system designed with Scade

The three blocks have their own clock: *cll*, *clc* and *clr*. Data transferred from a block to another one are delayed by means of the *fby* operator: as said in §3.1, the delay accounts for undetermined transmission delay. Relations between clocks are needed to take into account the periodicity of the three units: *assert Same_Period(cll, clr)* states that both clocks cll and clr have almost the same period. *Same_Period* is a Lustre node included in a library.

The proposed formalization provides a simple way to describe CPSP systems. Then, each block can be designed as a single component using the usual engineering practices.

4.3 Formal verification of properties

As said in §2.3.1, formal verification of properties involves defining the properties that the system should satisfy, and the assumptions on the environment behavior.

4.3.1 Properties

The program must verify the following properties:

- Safety: there must be no collision and no derailment. Collision occurs when two trains meet (if going in opposite directions) or reach (if going in the same direction) one another on the single line or on the same in_track, whereas derailment takes place if the physical path corresponding to the selected direction is not established while a train is moving.

- Fairness: there must be no starvation, i.e it should not be the case that two successive trains go in one direction while another train is waiting in the opposite direction.

4.3.2 Hypothesis

We consider the following hypothesis on the environment:

- The single line is initially not busy. It becomes busy only if a train leaves its in_track.
- Trains removed from their in_tracks will certainly pass on the single line.
- An edge of a switch is set only if it is controlled by the program.
- A switch can't be found in different positions at the same time by two sensors.
- The environment evolves much slower than the control program so that the control program can see all the changes of the inputs. In other words, clocks are fast enough to take samples of each level of each input.

4.3.3 Results

Formal verification with Lesar partly fails because of the state space explosion. The proof ran for 3 days: 1849225 states have been explored without showing the violation of the property but 4515433 states were expecting to be explored. We stopped the proof at this point since this scale of duration is not acceptable in an industrial context.

4.4 Automatic testing

As formal verification is not fully successful, it is interesting to test the distributed control program by means of the automatic testing method (§2.3.2). First, we look at the functional behaviour of the system assuming simplified clocks. Then, we focus on verification of properties.

4.4.1 Simulation of the system behavior

The automatic testing method can be used to simulate a specific behaviour of the system by defining assumptions corresponding to a particular context; thus, the designer can check that the system reacts as expected.

The first step of our experimentation aims at simulating the system behaviour in a restricted context: clocks of the three units are the same deterministic periodic signal as illustrated below:

cll	**t**	f	f	f	f	**t**	f	f	f	...
clc	f	f	f	**t**	f	f	f	f	**t**	…
clr	f	f	**t**	f	f	f	f	**t**	f	...

Hypothesis on the environment are preserved. The results are given on the chronogram of Fig. 7.

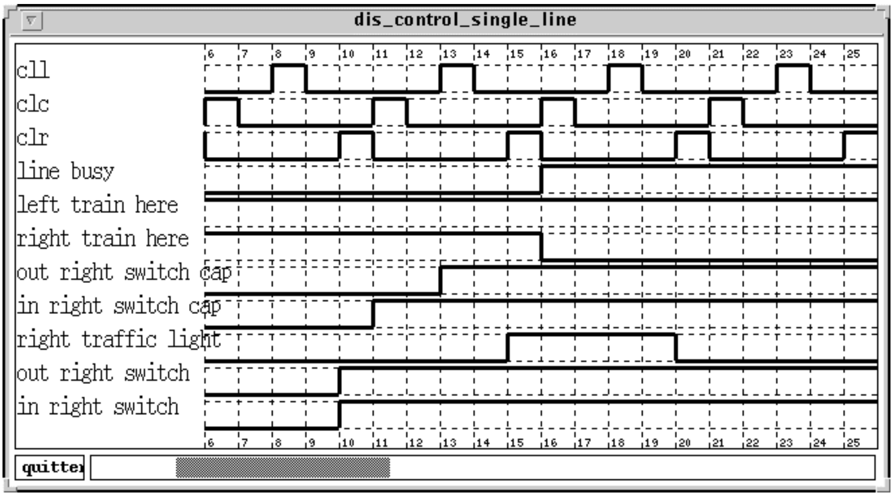


Fig. 7. Results from automatic testing

Let us detail steps from 10 to 20. Values of *cll*, *clc*, ..., *in_right_switch_cap*, are automatically generated by Lurette. Values of *right_traffic_light*, ..., *in_right_switch*, are responses provided by the control program.

1. Step 10: trains are detected on the left in_track and on the right in_track (the corresponding inputs —*left_train_here* and *right_train_here*— are *true*). The single line is empty (*line_busy* is *false*). The control program orders the switch to link the right in_track and the right out_track with the single line (the corresponding outputs —*out_right_switch* and *in_right_switch*—become *true*).
2. Then, the switches slowly move: at step 11, one links the right in_track with the single line (the input *in_right_switch_cap* becomes *true*), and at step 13, the other one links the right out_track with the single line (*out_right_switch_cap* becomes *true).*
3. Since the physical path is now ready, the right unit can allow the train to pass by controlling the right traffic light. This is done at step 15 when the right unit is activated (*clr* becomes *true*): access to the single line is granted (*right_traffic_light* becomes *true*). Then, the train leaves the right in_track (*right_train_here* becomes *false*) and enters the single line (*line_busy* becomes *true*).
4. When the right unit is activated again at step 20, it sets *right_traffic_lights* to *false* thus forbidding access to the single line, since it is busy.

This example shows that the designed system reacts as expected in a given situation. Of course, the properties that the system should satisfied are also examined: they are not violated by the generated test inputs. The next paragraph focuses on intensively testing the distributed program to check whether the properties are satisfied.

4.4.2 Observation of the properties

This experimentation involves the properties defined for formal verification purpose (§4.3.1). Ten test sequences which length is 100 have been generated: this length occurred to be a relevant length to observe trains moving on the single line.

The properties are never violated by the generated test inputs. Of course, this results does not mean that properties are proved. But since formal verification is not tractable, automatic testing is an alternative means to gain a sound confidence in the system safety.

5 Conclusion and future work

In this paper, we show that the considered distributed systems —Communicating Periodic Synchronous Processes— can be thoroughly formalized within the programming language Lustre-Scade thanks to the available sampling and holding mechanisms, and thanks to the assertion mechanism. This result yields valuable consequences:

- the same framework can be used for both programming, simulating, testing and proving properties of a distributed system.
- the Lustre-SCADE available analysis tools, e.g. the Lesar prover and the Lurette test generator, can be directly applied to distributed systems.
- this formalization is fully consistent with the usual engineering abstractions (period, delay) concerning smooth signals.

The next steps of our work are:

- apply the method to a real case study from Schneider Electric;
- try to improve the proof method so as to efficiently cope with these kind of Distributed Control Systems; alternative proof methods are also considered [15].
- go on with further research work on discontinuous signals: the proposed clock formalization works only for smooth signals. When it comes to discontinuous signal (boolean, integers,…) properties, it may be in many cases much better to solve the inverse problem, i.e. instead of distributing a program and then trying to check properties on the distributed program, take a synchronous program with already checked properties and try to safely distribute it while keeping these properties.

Acknowledgement

We wish to thank Christine Bodennec for her constructive comments during the preparation of this paper.

References

1. J. R. Abrial. *The B-Book*. Cambridge University Press, 1995.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183-235, 1994.

3. S. Bensalem, P. Caspi, C. Parent-Vigouroux and C. Dumas. A methodology for proving control systems with Lustre and PVS. *Proceedings of the 7th Working Conference on Dependable Computing for Critical Applications (DCCA7)*, San Jose, January 1999.
4. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270-1282, September 1991.
5. J.L. Bergerand and E. Pilaud. SAGA: a software development environment for dependability in automatic control. In *Safecomp'88*. Pergamon Press, 1988.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. P. Raymond. Recognizing regular expressions by means of data flows networks. In *23rd International Colloquium on Automata, Languages, and Programming (ICALP'96)*, Paderborn, Germany. LNCS 1099, Springer Verlag, July 1996.
14. 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.
15. M. Säflund and G. Stalmarck. Modelling and verifying systems and software in propositional logic. In *Proceedings of 17th IFAC Safecomp*, pp. 31-36, 1990.