Integration of Dependability Features in a Synchronous Application

Frédéric Boulanger[1]

Abstract – *We present here an overview of a new approach with associated tools, to implement dependability strategies for applications that follow the reactive synchronous approach. Starting from the description of an application as a graph of interconnected components, we model dependability policies as transformations of this graph. The transformed graph describes a new version of the application that integrates dependability features such as multiple copies of some components, voters that compare the outputs from copies of a component, or behavior checkers that compare the behavior of a component to an expected behavior. The graph transformations rely explicitly on the assumption that the components obey a synchronous execution model.*
The design of the dependability policies is not addressed. Our goal is only to provide dependability experts with a language for describing such policies and for integrating them into an application. The integration is done off-line and generates a new application with a structure that won't change at runtime. However, runtime changes in the structure of an application are possible and are discussed at the end of this article.
Keywords*: Architecture description language, Dependability, Model transformation*

I. Introduction

Dependability is a complex property that covers reliability, availability, safety and security [1]. When designing a system, dependability should be taken into account from the beginning [2], along with the functional aspects of the application. However, experts on the application domain of the system may not be experts on reliability, and a given system may be targeted to several implementations with different reliability requirements. Therefore, it is useful to consider dependability as a non-functional aspect of a system, and to weave dependability aspects into the design of the main functionality of an application. This paper does not address the design of dependability policies, nor the scheduling and the placement of the components on a given architecture, as discussed in [3]. Our approach provides means to describe dependability policies and to apply them to an application. Therefore, what is proposed here is not an aspect-oriented approach to dependability as presented in [4], but a tool for integrating dependability features into an application.

In our component based approach, an application is considered as a set of components that communicate through input and output signals using a synchronous execution model. The use of a synchronous execution model is essential because it guarantees the consistency of the flows of data between the components. Moreover, a subset of components that interact in a synchronous way may be considered as a single synchronous component, allowing for hierarchical abstraction.

For instance, figure 1 shows the original graph of a simple application on the left, and the graph obtained after applying the dependability policy: "replace component M_2 by three copies of it and insert a voter to check the consistency of their outputs" on the right. In such a transformation, we rely on the synchronous execution model to guarantee that at each instant, components M_{21}, M_{22} and M_{23} receive the same input from M_1 and that the voter works on samples that are computed from the same input.

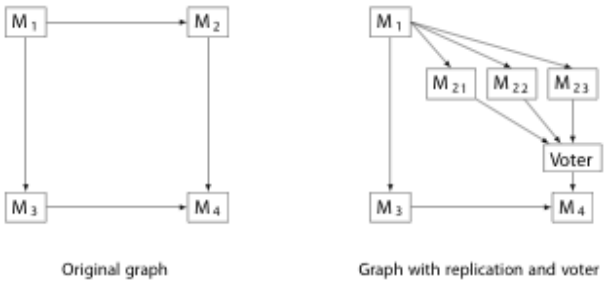

Fig 1. Example of graph transformation

If the execution platform is distributed, ensuring the synchronicity of the behaviors of the components has a cost. However, there exist efficient distributed execution techniques for synchronous systems as shown in [5].

Dependability features are described as transformations of the graph of an application. These transformations include: changing the connections between component signals, adding components, and replacing a component by a sub-graph that has the same inputs and outputs. This approach is related to the approach used in the Hydra tool [6][7], with the

constraint that the components obey a synchronous model of execution. However, our approach supports more general transformations of the application than the replication of components.

II. BDL: a Block Description Language

The description of the graph of an application relies on the description of the interface of its components, or blocks, and on the description of the connections between the signals of these blocks. In our approach, these descriptions are made in BDL, short for Block Description Language, an architecture description language designed specially for this purpose. This language can be compiled into C++ to produce an executable application. The choice of C++ was made because we rely on object-oriented features to instantiate and connect blocks, and because it is possible to bind code written in other programming languages like C, Fortran or ADA in C++. Each block used in an application must therefore be encapsulated in a C++ class. We have developed tools to produce such classes from Esterel [8] modules and Lustre [9] nodes. Runtime support for the synchronous execution model assumed by BDL is provided by the libSync library that was developed in previous work [10].

II.1. Description of a block

For each block, BDL gives:

- the source of the description of the block, which may be created manually or generated automatically from the source code of the block;
- the name of the C++ file that contains the class of the block;
- the definitions used by the class (types, other classes, libraries);
- the parameters of the block if any;
- type equivalences that match types in different programming languages or operating systems;
- the interface of the block: its inputs and outputs, with their types. An output may be declared as not depending instantaneously on the inputs of the block. This property is used to check that loops in the graph of an application are not instantaneous and that the activation of the blocks can be scheduled.

For instance, let us consider a block that checks the equivalence of two pure signals `inp1` and `inp2`. When one of the input signals is present while the other is not, the block emits an output named `failure`. As soon as `failure` has been produced, another output named `has_failed` is emitted at each instant. The interface of such a block could be described as:

```
#source "CheckEQUIV.strl"
#c++ "CheckEQUIV.H"
CheckEquiv {
  input:
    inp1,
    inp2;
  output:
    failure,
    has_failed;
}
```

This description declares a block named `CheckEquiv` and states that this block is defined in Esterel in the source file CheckEQUIV.strl, and that the `CheckEQUIV` C++ class is declared in the file CheckEQUIV.H. This block has two pure (not valued) inputs named `inp1` and `inp2` and two pure outputs named `failure` and `has_failed`.

II.2. Block templates

Most dependability features depend only on the topology of the application, not on the exact type of the data that goes from a block to another. To avoid the need to declare a specific version of a dependability block for each type of data, BDL supports templates, or parameterized blocks. The mechanism and the syntax are borrowed from C++, and the parameters of a block may be either types or values. For instance, one can describe a comparator that compares any number of inputs of the same type and tells whether they are equal or not as follows:

```
Comparator<T><int N> {// N inputs of type T
  input  : T[N]       // array of inputs
  output : boolean equal; // boolean output
}
```

Then, if we need to compare 10 integers, we can just declare:

```
Comparator<int><10> a_ten_int_comparator;
```

II.3. Description of an application

An application is a set of interconnected blocks. It is therefore itself a block and its interface is described in BDL in the same way as for a block. After the description of the interface comes the declaration of the blocks that compose the application, and then the connections between the inputs and outputs of the blocks. The inputs and outputs of the application are aliased to inputs and outputs of blocks of the application. The difference between the connection of an input to an output and the aliasing of two signals is that in the former case, one signal (the output) produces data for the other signal (the input). Aliasing two signals just states that a signal of the application is indeed a signal of one of its block. Therefore, it is only possible to alias an input to an input and an output to an output.

Let's consider an ABS brake system composed of a brake and an ABS controller. The brake and the ABS controller are described as follows:

```
Brake {
  input : int command;  // user input
  output: int pressure; // hydraulic command
}
ABS {
  input : int raw_cmd;  // raw  command
  input : skidding;     // tire is skidding
  output: int ref_press;// regulated command
}
```

The complete ABS brake system is built from a brake and an ABS controller:

```
#use Brake    // include description of brake
#use ABS      // and of ABS controller

ABSBrake {
  input : int command;  // user input
  input : skidding;     // tire is skidding
  output: int pressure; // hydraulic command

  Brake brake;      // an instance of Brake
  ABS   controller; // and of ABS controller

  // the input of the brake is the 'command'
  // input of the application
  brake.command = command;
  // the 'skidding' input of the controller
  // is the one of the application
  controller.skidding = skidding;
  // the output of the application is
  // the output of the ABS controller
  pressure = controller.ref_press;

  // connect the 'raw_cmd' input of the
  // controller to the output of the brake.
  controller.raw_cmd << brake.pressure;
}
```

The alias (=) operator is used to alias signals, while the connect (<<) operator is used to connect an input to an output.

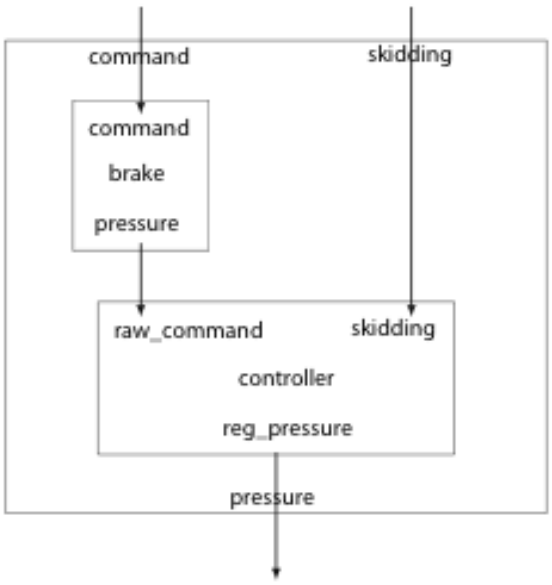

Fig. 2. Structure of the ABS brake system

The structure of the resulting ABS brake system is shown on figure 2.

III. Dependability statements

The syntax for describing blocks and connecting them to build applications or new blocks has now been presented, and the following shows how BDL allows a designer to modify the topology of an application to implement dependability policies.

BDL has operations for:

- adding new blocks and signals to an application,
- replacing blocks,
- changing the connections and aliases between signals,
- getting information about the structure of the application,
- controlling how operations are performed.

III.1. Adding blocks or signals

The implementation of a dependability policy may require that new inputs or outputs are added to the application, for instance when hardware redundancy is used. New signals are declared using the `input:` and `output:` statements, like in the description of a block. In the same way, new blocks can be added to the application by declaring them with the same syntax as in the definition of a block.

However, a dependability policy is implemented as a procedure and may be applied in different contexts where the names of the new signals and blocks are not known when the procedure is defined. Therefore, BDL has a `##` binary operator that builds names by concatenating its first argument (which must be an identifier) and its second argument (which may be an identifier or an integer). This allows the use of syntactical arrays, which are not actual arrays, but sets of identifiers that are built from consecutive integers. Syntactical arrays have the advantage that they can contain objects of different types, contrary to real arrays that are homogenous, and that each item of the array can be initialized individually. This is of particular interest for arrays of voters since each voter has a type that depends on the type of the signals it processes.

III.2. Replacing blocks

An important operation when transforming an application is the replacement of a block by a subgraph of interconnected blocks. This operation is transparent for blocks that were connected to the replaced block because the subgraph which replaces the block has the same interface and is connected in the same way as the original block. However, the replacing graph may have additional signals to exchange diagnostic information with other parts of the application.

The syntax for replacing a block is:

```
someblock #becomes someproc(arg1, arg2, ...)
```

where `someblock` is the block to replace, and `someproc` is the procedure that builds the subgraph that replaces the block. The practical uses of `#becomes` will be exposed later.

The connections between blocks can be changed using the connect (<<) and alias (=) operators. When expressing transformations of an application, the same signal can be aliased or connected several times: only the last alias or connection is taken into account. This allows the use of generic procedures that apply a global policy to all signals, followed by special processing of some signals.

III.3. Getting information about the application

In order to transform the graph of an application, it is necessary to get information about its structure. The following operations can be applied to a block:

- `type()` yields the type of the block. This type can be used to declare other blocks of the same type;
- `nbInputs()` returns the number of inputs of the block;
- `nbOutputs()` returns the number of outputs of the block;
- `inputs(i)` returns the i^{th} input of the block;
- `outputs(i)` returns the i^{th} output of the block;

The following information operations can be applied to a signal:

- `type()` yields the type of the signal;
- `source()` yields the signal to which an input is connected (it can be applied to inputs only).

The source of an input is generally an output. However, if the input is aliased to an input of the application, the source is an input. When defining dependability policies, the context in which they will be applied is not known yet, so the designer doesn't know whether `source()` will return an input or an output. To solve this issue, the behavior of the << and = operator is special when their second argument is of the form `signal.source()`: << behaves like = if the source is an input, and = behaves like << if the source is an output.

III.4. Control statements

BDL has statements to control the application of operations and procedures:

```
#iterate(var, bound) {
  commands;
}
```

executes `commands` with `var` varying from 1 to `bound`. The `var` variable is local to the block delimited by `{` and `}`.

```
#if(type1 ~ type2) {
  commands_if_same_type;
}{
  commands_if_different_types;
}
```

checks if `type1` and `type2` are equals. The special value `0` can be used to denote the type of pure signals. Pure signals carry events that don't have a value.

```
#if(int1 = int2) {
  commands_if_equal;
}{
  commands_if_different;
}
```

checks for equality of `int1` and `int2`.

III.5. Dependability policies

Dependability policies are defined using primitive statements of BDL and other dependability policies. The syntax of the definition of a dependability policy is:

```
#def policy(arguments) {definition};
```

As an example, let us consider the definition of a policy that replaces a block by n copies of this block, with voters to merge the outputs of the copies:

```
#def nplication(block, n) {
  CopyInputs(block);
  // declare n instances of the block
  #iterate(i, n) {
    block.type() B##i;
  }
  // add a voter for each output
  #iterate(i, block.nbOutputs()) {
    #if(block.outputs(i).type() ~ 0) {
      // pure voter with n inputs
      PureVoter<n> V##i;
    }{
      // typed voter with n inputs
      Voter<block.outputs(i).type()><n> V##i;
    }
    // connect voter inputs
    #iterate(j, n) {
      V##i.inputs[j] << B##j.outputs(i);
    }
    // alias voter output
    block.outputs(i) = V##i.output;
  }
}
```

Such a policy may be used to replace a block with 3 copies of this block:

```
block #becomes nplication(block, 3);
```

as shown on figure 3. The first statement calls a procedure to insert blocks that copy their input to their output. This is necessary because we can alias only one signal to a signal of the application. Since the three copies of the block must read their inputs from the inputs of the original block, we have to copy these inputs in case they are inputs of the application. The synchronous execution model of the application allows the insertion of such blocks without changing the behavior of the application because the outputs of the blocks are available at the same instant as their inputs. The same property is used to guarantee that the voters do not introduce a delay in the processing of the outputs of the copies of the replicated block.

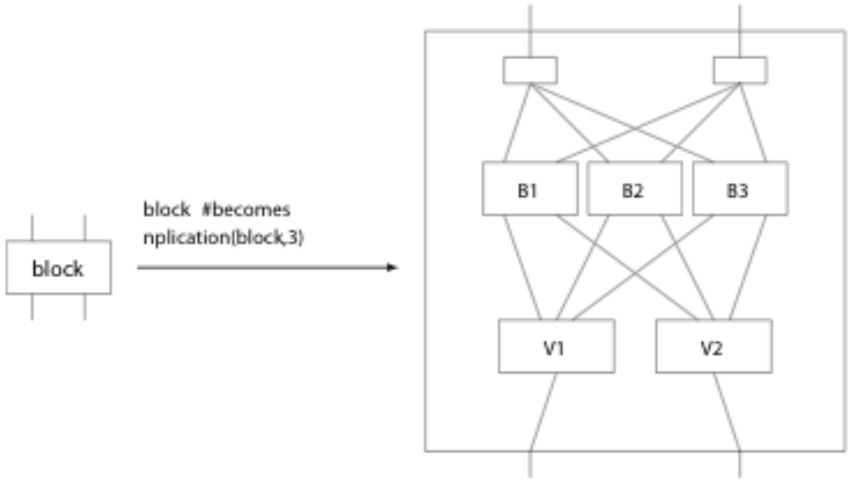

Fig. 3. Replication of a block

IV. The synchronous execution model

The BDL language, and the associated BDLC compiler which translates the description of an application and the dependability policies into C++, rely on a synchronous execution model to ensure the consistency of signals in the transformed application. This execution model assumes that:

- the blocks of an application communicate through signals, there are no shared variables;
- there is a notion of logical instant, shared by all the blocks of an application, and at each instant, the value and status of interconnected signals is the same;
- blocks react instantaneously: they produce their outputs at the same instant as they read their inputs.

This execution model allows us to replace a block by a set of interconnected blocks without modifying the synchronization of data flows in the application because inserting a block along the path between an output and an input does not introduce any delay in the processing of data. It also guarantees that multiple copies of a block receive the same data at the same instants, so that their behaviors can be compared safely, any discrepancy in their outputs being due to a failure in their processing, and not to the fact that they work on different inputs.

However, this implies that any loop in the graph of the application is instantaneous. For such a loop to be causal, there must be, at each instant, a unique value of the signals that obeys the semantics of the blocks. Since BDL considers the blocks of an application as black boxes, their semantics is unknown, and the causality of an instantaneous loop cannot be checked. Therefore, instantaneous loops are forbidden in BDL, and the BDLC compiler checks for instantaneous loops, as well as for unconnected signals.

Non instantaneous loops are allowed, but for BDLC to see that a loop is not instantaneous, the pure "black box" approach must be relaxed to say which outputs of a block do not depend instantaneously on its inputs. This is done with the `#nodep` statement, as illustrated by the following description of a delay block:

```
#c++ "Delay.H"
// a one tick delay for signals of type T
Delay<T> {
  input : T;
  output : T;
  #nodep output;
}
```

In a delay, the output depends only on the state of the block. The current input gives the next output, not the current one, so the output does not depend instantaneously on the input of the delay.

IV.1. Clocks

In an application, each block belongs to a clock which defines the instants at which the block will react to its inputs and produce its outputs. There may be several clocks in an application, one for each connected subgraph of the application. For each clock, BDLC computes the partial order induced by the connections between signals and computes a schedule (a total order of activations) that is compatible with this partial order. Several schedules may be compatible with the causal partial order, and all should yield the same results because the value and status of a signal are unique at each instant: a signal cannot be seen absent and then present in the same instant. However, if some blocks have side effects, like writing data to a file, the order in which blocks are activated at each instant may influence the observable behavior of the application.

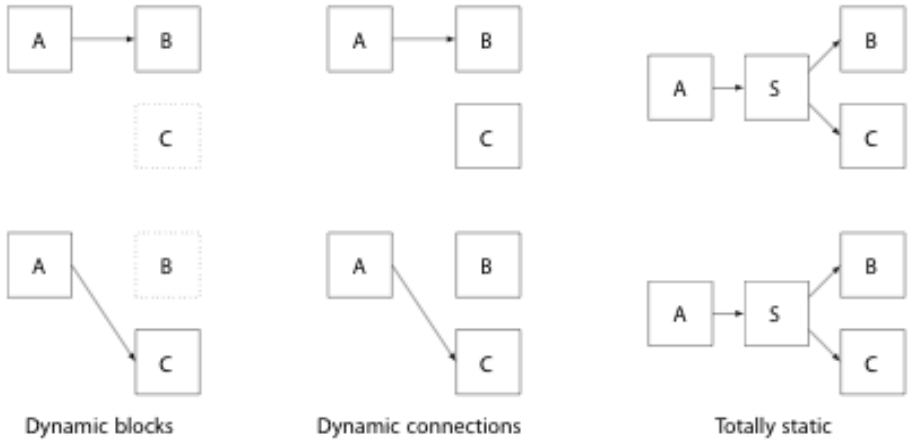

Fig. 4. Reconfiguration methods

One possibility to solve this issue is to add dummy inputs to some blocks and to connect them to the outputs of the blocks after which they should be activated. These dummy connections will turn the partial order into a more complete order that is compatible with the expected behavior of the application, side-effects included. These dummy connections are a hand-coded representation of the dependencies between the side effects. A better solution is to have only side-effect free blocks, or, if necessary, to group all side-effects in one block which processes them in the right order.

IV.2. Asynchronous communication

Blocks which belong to the same clock communicate synchronously, but they must be able to communicate asynchronously with the external world, or with blocks which belong to other clocks and therefore do not share the same logical instants. Communication from a clock toward the asynchronous outside is not a problem: a synchronous event becomes an asynchronous event when observed from the outside of the clock. However, creating a synchronous event from an asynchronous one may be more difficult.

The simplest solution is to memorize the occurrence of an asynchronous event when it occurs, and to generate the corresponding synchronous event at the next instant of the clock. However, this solution may not give a coherent view of the asynchronous world to the synchronous clock. For instance, let's consider two asynchronous events: `ms` occurs every millisecond, and `s` occurs every second. From a synchronous point of view, each time `s` occurs, `ms` should also occur at the same instant since every second is made of a whole number of milliseconds. However, non-deterministic processes in the implementation of an application may lead to an occurrence of `s` with no simultaneous occurrence of `ms`. In this case, when `s` is detected, a synchronous event for `s` should be produced at the next instant where `ms` has also been detected. Moreover, the thousandth occurrence of `ms` should be simultaneous with an occurrence of `s`.

As we can see, building synchronous events from asynchronous ones depends on the semantics of the signals and cannot be done automatically. In our execution model, synchronous events are built from asynchronous ones by interface blocks. Such blocks can communicate with a clock to create new instants when needed. Therefore, synchronous clocks can be built from the stimuli of the application as well as from a periodic activation loop.

IV.3. Dynamicity

BDL has no support for dynamicity. It rewrites the graph of an application statically, at compile-time, what makes it possible to check for instantaneous loops, unconnected signals and so on. However, the synchronous execution model used to execute the application supports dynamicity. It is possible for a block, as part of its reaction to an event, to destroy or create other blocks and to change the connections between signals. These changes are not instantaneous: they are applied after all the components that belong to the same clock have reacted to the instant when the change is requested. See [11] for a discussion on the reconfiguration of data-flow models. Such dynamic reconfigurations of an application force the clocks to recompute a schedule for their blocks, and may lead to a runtime exception if the new configuration of the application cannot be scheduled.

Dynamicity may be useful to optimize the use of the available resources of the execution platform. By creating blocks only when they are needed, one can reduce the memory footprint of an application. However, the risk of running into a lack of memory when a block must be created, or to fall into a configuration that has not been statically checked for correction is often too high to be worth the gain in statically allocated resources.

Figure 4 shows three methods to handle the reconfiguration of an application, from the most dynamic on the left to the totally static approach used by BDLC on the right. The first method goes from the upper configuration to the lower by destroying block B, creating block C and reconnecting the input of C to the output of A. The second method, in the middle, goes from the upper configuration to the lower one by disconnecting the input of B from the output of A, and connecting the input of C instead. With this method, blocks B and C exist during the whole lifetime of the application.

With the last method, on the right, no block is destroyed or created and no connection is changed. In the upper configuration, the switch block S copies its input to its upper output toward B. In the lower configuration, it copies its input to its lower output toward C. The main drawback of this method is that the graph of the application does not show explicitly that only one of B and C is active at any time.

V. Conclusion

By considering an application as a set of interconnected black boxes that obey a synchronous reactive model, BDL allows the description of dependability policies as transformations of the graph of the application. The synchronous execution model makes these transformations safe because blocks react instantaneously to their inputs, and signals have a unique value at each instant, which is propagated instantaneously along connections. Therefore, introducing new blocks in a flow of data does not delay the data.

This approach releases the tight coupling between functional properties of an application and dependability, and makes it possible to produce several implementations of an application by applying different dependability policies to the initial description of the application. It can be considered as an aspect-oriented approach to dependability.

Several dependability blocks, like voters, behavior checkers and behavior predictors have been implemented, along with the dependability policies that integrate them in the graph of an application. Interface blocks are used to make the application communicate with its environment.

Improving the dependability of an application should not change its functional properties. When all the components of an application are designed using a synchronous language, it is possible to use formal verification tools to check that critical properties that hold on the original application still hold on the dependable application.

Future works will address the issue of the placement of the blocks on different processing units. Today, the synchronization of processing chains that run on different processors is made using interface blocks that implement inter-processor communications.

References

[1] A. Avizienis, J.C. Laprie, B. Randell and C. Landwehr, Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1), pp. 11-33, 2004

[2] A. Bondavalli, M.D. Cin, D. Latella and A. Pataricza, *High-level Integrated Design Environment for Dependability*. Fifth International Workshop on Object-Oriented Real-Time Dependable Systems, WORDS 1999, Monterey, USA, November 1999

[3] A. Girault, H. Kalla and Y. Sorel, A Scheduling Heuristics for Distributed Real-Time Embedded Systems Tolerant to Processor and Communication Media Failures. *International Journal of Production Research*, 42(14), pp. 2877–2898, July 2004

[4] R. France, I. Ray, G. Georg and S. Gosh, Aspect-oriented approach to early design modeling. *IEE Proceedings - Software*, vol. 151, pages 173–185, August 2004

[5] P. Caspi, A. Girault and D. Pilaud, *Distributing Reactive Systems*. Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94, Las Vegas, USA, October 1994

[6] P. Chevochot and I. Puaut, *Scheduling Fault-Tolerant Distributed Hard Real-Time Tasks Independently of the Replication Strategies*, 6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA '99), Hong Kong, China, December 1999

[7] P. Chevochot and I. Puaut, *An Approach for Fault-Tolerance in Hard Real-time Distributed Systems.* 18th IEEE Symposium on Reliable Distributed Systems, Lausanne, Switzerland, October 1999

[8] G. Berry, *The foundation of Esterel* (MIT Press, Robin Milner edition, 1998)

[9] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, vol. 79, nr. 9. September 1991

[10] G. Vidal-Naquet and F. Boulanger, *Integration of Synchronous Modules in an Object-Oriented Language*. Information Systems - Correctness and Reusability, selected papers from the IS-CORE Workshop, editors: R.J. Wieringa and R.B. Feenstra, World Scientific 1995, pages 279–291

[11] S. Neuendorffer and E.A. Lee, *Hierarchical Reconfiguration of Dataflow Models*. Invited paper, Conference on formal methods and models for codesign, MEMOCODE, San Diego, California, June 22-25 2004

Authors' information

[1]Supelec – Département Informatique, France.

Frédéric Boulanger is a professor at Supelec, a major French grande école. He got his engineering degree from Supelec in 1989, and a PhD in Computer Science from Paris-Sud University in 1993. His current interest is in heterogeneous modeling and the precise definition of the interactions between models of computation.