INTEGRATION OF DEPENDABILITY MODULES IN A REACTIVE APPLICATION

Frédéric Boulanger, Guy Vidal-Naquet

Supélec – Service Informatique
Plateau de Moulon
Gif-sur-Yvette, France

and

Laboratoire de Recherche en Informatique
Université de Paris Sud
Orsay, France

ABSTRACT

The design of control-command applications can lead to clusters of reactive modules that communicate synchronously. They can be considered as a graph whose nodes are the modules and whose arcs are the communication paths.

Moreover, if each module is an object with encapsulated data and methods for self duplication and connection to other modules, it becomes possible to modify the application by "pruning" and "grafting" modules in the graph.

The synchronous nature of communications in the graph ensures that a given data sent to several modules through different paths will get to all its destinations at the same time.

We have developed a language that allows us to modify the graph of such applications. Sets of modifications can be grouped in parameterizable macro-definitions. It is possible to use this language to apply operations that enhance dependability, such as the replication of modules, the insertion of voters or property checkers. These operations are applied at the level of the description of the application, and it is possible to generate different applications from the same original design according to different dependability constraints.

INTRODUCTION

Developing applications with dependability constraints generally requires that the developers are specialists of both the application domain and dependability techniques. Moreover, a global approach makes it difficult to distinguish what stems from the application from what has been added to make it reliable. A small change in the dependability constraints can therefore lead to a complete rewriting of the application, and a small change in the application can bring a complete change in the dependability mechanisms.

This leads us toward an approach of design where the application and the dependability mechanisms are not completely coupled, with four main advantages:

- application developers can concentrate on their domain, leaving dependability issues to other specialists,
- changes in the application and change in the dependability mechanisms can be considered separately, and will influence each other only in the integration phase,
- dependability mechanisms can be stored as libraries and reused in several applications,
- several applications can be obtained from the same original design by applying different set of dependability macros. This can also be used to handle several configurations for the same application.

In order not to loose the time saved during the development when we integrate the dependability mechanisms into the application, the integration process must be automated. This requires some properties of the description of the application:

- the integration must be independent from the semantics of the application, which will therefore be described as a set of interconnected black boxes.
- the semantics of communications between these boxes must be very simple. Since the integration of the dependability mechanisms will add layers of modules in the application, the value of data going along a path must be independent from the number of layers this path goes through.

The last property is true in the synchronous reactive model of communication since it assumes instantaneous broadcast of values in the application.

Our model of an application will be a graph of interconnected black boxes that communicate and compute instantaneously. In an effective implementation this will not be true, but this hypothesis can be made at the model level and will be verified in logical time. The integration of the dependability modules will consist in a modification of the original graph with the warranty that data coherence will not be broken by the insertion of the new modules.

The following simple example shows the steps that lead from an initial application to its dependable version:

1. from the description of an application built from interconnected synchronous modules, we built the interconnection graph of the application as shown in figure 1;

2. following dependability policies determined from a specific analysis, we modify the original graph to obtain the graph of the dependable application as shown in figure 2. Here, we assume that the followed policy is "make three copies of M2 and vote on the outputs of the copies";

3. from this new graph, we generate the code of the dependable application.

This approach may be used to obtain other types of properties from an application. For instance, it is possible to develop macros that add confidentiality properties to an application, as shown in figure 3. In this small example, each message is scattered on three communication channels so that the knowledge of what goes through one channel cannot lead to the original message contents.

NATURE AND PROPERTIES OF THE MODULES

The modules used to build the application can be written in any programming language, but they must have certain properties to allow the integration tool to rearrange them into a new application.

Building the graph that represents the application requires the knowledge of the interface of the modules, how they are connected, and the type of the data they exchange.

We recommend the use of synchronous languages (like Esterel, Lustre and Signal) because the mathematical semantics of these languages makes it possible to check critical properties of the modules. The synchronous hypothesis (assuming that computation and communication take no time) eliminates the issue of keeping data coherence between the original and the final application. Let us consider the example of figure 2: we must be sure that modules M21, M22 and M23 work on the same data, and that the voter compares outputs corresponding to the same inputs of the modules. This is true when the synchronous hypothesis holds.

The synchronous model is well suited to the hierarchical design of a system since a group of interconnected synchronous modules can be considered itself as a single synchronous module. We can therefore describe an application at several levels of abstraction, just as we can do in the object oriented model that we use in the implementation.

The operations that must be supported by the modules are easy to provide in an object oriented model: reuse of components, interface declaration, copy of objects. The modules will be written in an object-oriented language, or an object-oriented language will be used to embed their code and give them the required "object properties".

OVERVIEW OF THE TOOLS: AN EXAMPLE

To validate our approach, we have designed a language, ADML (Application Description and Modification Language) which is used both to describe an application and to express the operations on this description.

A compiler, ADMLC, reads the description of an application and a set of operations to execute, and produces the description of the new application. It also produces the C++ code of the new application.

The description of a module contains five parts:

- the C++ class that implements the module,
- the definitions used by this class,
- the parameters of the module,
- the type equivalences, to map types from one language to another,
- the interface of the module: name, type and dependency properties of the inputs and outputs.

We will not give here the complete description of ADML, but rather show its main features through an example.

We consider a very simple module that controls a valve that regulates pressure: if the pressure is higher than a given threshold during a given amount of time, it should open the safety valve.

```
#c++ ValveControl.h
#param {
  float threshold;
  int duration;
}
ValveControl {
  input: float pressure;
  output: valve;
}
```

The module takes the threshold and the amount of time as parameters, receives the pressure as a floating point value, and drives the valve through a boolean output.

Let us build a small application with a captor, a valve and our control module: all we have to do is connect the output of the captor to the input of the control module, and connect the output of the control module to the input of the valve.
We assume that we have the following declarations:

```
PCaptor {
  output: float pressure;
}

Valve {
  input: command;
}
```

We can then write the description of our application as follows:

```
#use Valve
#use PCaptor
#use ValveControl

Application : {
  PCaptor captor;
  Valve   valve;
  // Pressure threshold is 10e6 Pascal
  // Duration is 2 seconds max.
  ValveControl control(10e6,2);

  // Now, we make the two connections
  control.pressure << captor.pressure;
  valve.command << control.valve;
}
```

Notice that no information is given on what the different modules really do. We just specify which modules we use, how they are connected, and the type of the data they exchange.

Suppose now, that the dependability expert determines we must have three copies of the control module, and the third copy must take its input from a new captor. The valve will be driven by a voter that will look at the output of the three control modules.

As a first step, we will create the three control modules and add the voter. Then, we will connect the third control module to a new captor. We can define a generic replication macro in ADML:

```
#def replication(module, number) {
  #iterate(i, number) {
    module.type() module_##i;

    #iterate(j, module.nbInputs()) {
      module_##i.inputs(j)
           << module.inputs(j).source();
    }
  }
  #iterate(i, module.nbOutputs()) {
    Voter<module.outputs(i).type()><number>
          module_##vote_##i;
    #iterate(j, number) {
      module_##vote_##i.input
           << module_##j.outputs(i);
    }
    module.outputs(i)
         = module_##vote_##i.output;
  }
```

Basically, this macro creates `number` objects of same type as object `module` and names them by appending (with operator ##) the number of the object to the original module name. So if we apply this macro to our valve control module to build three copies of it, they will be named `control_1`, `control_2` and `control_3`.

Then, each input of each new module is connected (operator `<<`) to the output that drove the corresponding input of the original module (operator `source()`). We have now to add a voter for each output. Voters are generic modules: a `Voter<T><N>` votes on `N` inputs of type `T`. We connect each input of the voter to the corresponding output of the duplicated modules, and we replace the outputs of the original module by the outputs of the voters.

Applying `replication(control,3)` to our original application yields a new application as shown in figure 4.

The definition of this macro shows some important aspects of ADML that we are going to discuss briefly.

Since ADML is used to both describe and modify application, it has primitives to declare and connect modules, but also to get information about the current state of the application.

It is possible to get the type of a module or signal, to get the number of inputs and outputs of a module, or to know from where a module gets its inputs.

New names can be constructed with the ## concatenation operator and primitives such as `#iterate` or `#if` makes it possible to write generic macro-definitions whose behavior depends, for instance, on the type of a module or on the number of its inputs.

Another important feature of ADML is genericity: it is possible to describe generic modules such as voters or delays which implement the same behavior for different types or number of inputs/outputs. Code generation for such modules relies on genericity mechanisms in the target language. For C++, we use template classes.

The next step in our example is to add a new captor that will drive the input of the third control module with the following ADML code:

```
PCaptor captor2;
control_##3.pressure << captor2.pressure;
```

This yields the application shown in figure 5.

Note that if the environment of the application changes, and that five copies are needed instead of three to reach a more stringent set of dependability constraints, the new application can be obtained very easily by applying `replication(control,5)` to the original application.

We cannot describe all the features and syntax of ADML in this paper, but we hope that this example will give an idea of what is possible to do with it.

EXECUTION MODEL

The execution model that gives the semantics of our application graphs is synchronous reactive data-flow, which means that:

1. communication between modules is done by flows of events,
2. there is a global notion of instant, and events have the same meaning for all modules at a given instant and no meaning outside an instant,
3. a module produces its outputs at the same instant it receives its inputs. This hypothesis that reaction takes no time (in logical time) is known as the "Synchronous Hypothesis".

Choosing a synchronous execution model eliminates data coherence issues between several sources of information. For instance, if we want to write a time display module that receives hours, minutes and seconds from different sources, we have to be very careful to avoid the display of `8:59:59` just after `7:59:59` in an asynchronous execution model. With a synchronous execution model, the three inputs change at the same instant, and the display module cannot see the hour changing just before the minutes or the seconds, even if it is what really happens in the implementation: the three inputs will be available only when they will have the right value for this instant.

The main application domain of the reactive synchronous model is exactly the one addressed by ADML: critical control software, where dependability issues are of paramount importance, and determinism a key issue.

However, there is a drawback to this approach: any loop in the data-flow graph is instantaneous. An instantaneous loop is causal if at each instant, there is a unique value of the data flows that respects the semantics of the modules. An application that contains causal instantaneous loops has a perfectly defined semantics. However, since we consider modules as black boxes, and cannot access there semantics, we must reject all instantaneous loops, assuming they are not causal.

If causal instantaneous loops cannot be eliminated from an application, they must be handled by tools such as synchronous language compilers that have access to the semantics of the modules and can generate correct code for such loops.

ADML allows non instantaneous loops, i.e. loops that contain a delay. Delays, or more generally, the fact that an output does not depend on the current reaction of a module, are indicated by the `#nodep` keyword. This allows the programmer to give the information needed by ADML to determine that a loop is not instantaneous, without going into the details of its semantics.

CONCLUSION

The tool presented here offers help at the design and code generation levels. It is not a tool for the analysis of dependability properties of an application, but it enables the automatic realization of dependability policies.

The automatic code generation it provides reduces the cost of changes in the dependability policy, and allows a same application to be easily adapted to several environments.

It is also interesting to note that from a given simple application, the integration of dependability modules can lead to very complex topologies, and that our tool ensures the correctness of the links between modules and of the types of the data they exchange.

Our tool is based on two abstractions:

- abstraction of the modules which are considered only through their interface. This abstraction is provided by the object-oriented approach,
- abstraction of the communications between modules which are considered to be instantaneous. This abstraction is provided by the reactive synchronous approach.

The integration of these two approaches in one model is the key feature of ADML and can be applied to many aspects of the development of complex systems.

We are studying its use for the separate development of the control and data processing parts of an application. Control is generally the most complex and the fastest evolving part of an application. Making it as little dependent as possible from the data processing part would ease the use of libraries of reusable components.

REFERENCES

Harel, D. and Pnuelli, A., 1985, “On the Development of Reactive Systems”, Weizmann Institute of Science, Rehovot, Israel.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D., 1991, “The Synchronous Data Flow Programming Language LUSTRE”, *Proceedings of the IEEE*, 79(9), September 1991.

Benveniste, A., and Le Guernic, P., 1990, “Hybrid Dynamical Systems Theory and the SIGNAL Language”, *IEEE Transactions on Automatic Control*, 35(5), May 1990.

Berry, G., and Gonthier, G., 1992, “The ESTEREL synchronous programming language: Design, semantics, implementation”, *Science of Computer Programming*, 19(2):87-152, November 1992.

Berry, G., Ramesh, S., and Shyamasundar, R.K., 1993, “Communicating Reactive Processes”, *Proceedings of the 20th ACM Conference on Principles of Programming Languages*, Charleston, Virginia.

André, C., Boulanger, F., Péraldi, M.-A., Rigault, J.-P., Vidal-Naquet, G., 1997, “Objects and synchronous programming”, *European Journal of Automation*, 31(3):417-432, 1997.

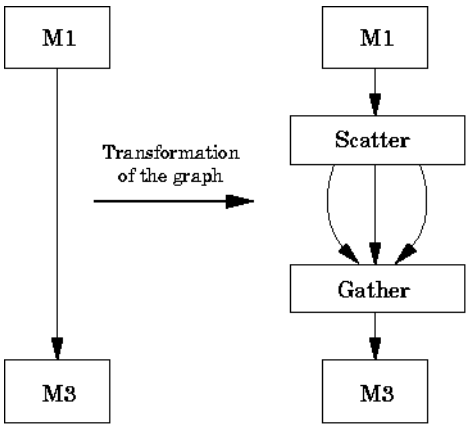

Fig. 3: Application to confidentiality

FIGURES

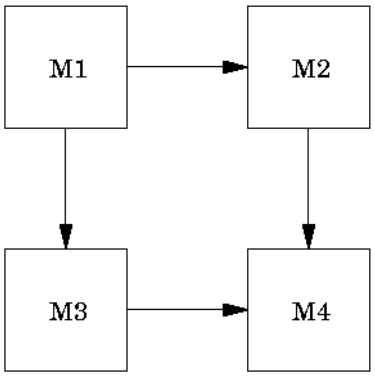

Fig. 1: Initial graph of the application

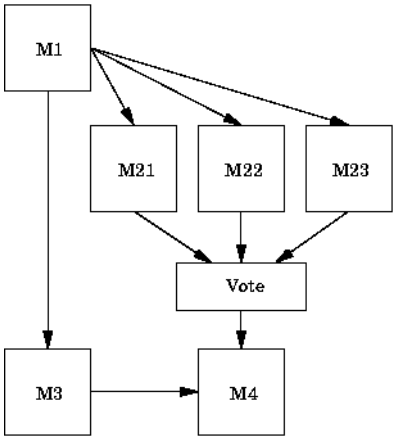

Fig. 2: Graph of the dependable application

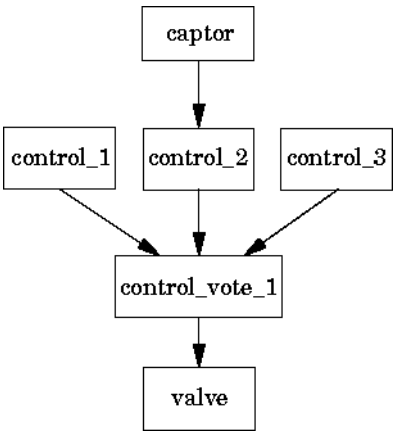

Fig. 4: Effect of the `replication` macro

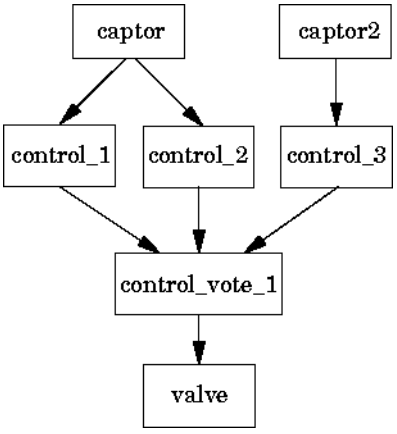

Fig. 5: Final application with 2 captors