Running Modes in a Dataflow Language: Mode-Automata and their Implementation*

Florence Maraninchi Yann Rémond
Yannick Raoul
VERIMAG†– Centre Equation, 2 Av. de Vignate – F38610 GIERES
http://www-verimag.imag.fr/PEOPLE/Florence.Maraninchi

1 Introduction

We are interested in *reactive* systems, which interact continuously with their environment. The *synchronous* approach [BB91, Hal93] to the programming of reactive systems is represented by imperative languages like Esterel [BG92] and Argos [Mar92], or by declarative dataflow languages like Signal [LGLL91] or Lustre [CHPP87, HCRP91]. In the field of reactive system programming, engineers who have to design control laws and their discrete form were used to block-diagrams. Lustre and Signal offer a structure and even a graphical syntax similar to that of block-diagrams. They have a formal semantics and can be efficiently compiled into C code, for instance. Lustre has been defined and implemented at the Verimag laboratory. Verilog S.A. sells a commercial programming environment based on Lustre; it is called SCADE, for *Safety Critical Application Development Environment.* SCADE is used for operational projects by Aerospatiale, Schneider Electric, Honeywell. The Verimag laboratory also develops a Lustre compiler, together with verification and testing tools. Evolutions of the language are first implemented using this academic platform, evaluated on case studies provided by SCADE users, and then transferred to SCADE.

Recently, the users of the SCADE environment expressed their need to specify part of a design as a state graph. Discussions about typical examples they had, led us to the following conclusion : there is a need for the expression of *running modes* in Lustre — and it would be the case for any other dataflow language. Stategraphs were used, more or less, in order to represent the mode structure of the system.

So, first, what is a “running mode”, and why do we feel we need a specific construct for describing modes in a dataflow language?

One (and perhaps the only) way of facing the complexity of a system is to break it down into several “independent” tasks. The tasks correspond to some abstractions of the global behavior of the system, and they may be viewed as different parts of this global behavior, devoted to the treatment of distinct situations. Decomposing a system into tasks allows independent reasoning about the individual tasks. Tasks may be *concurrent*, in which case the system has to be decomposed into concurrent and communicating components. The interface defines how the components communicate and synchronize with each other in order to reach a global goal. Thinking in terms of *independent modes* is in some sense

*This work has been partially supported by Esprit Long Term Research Project SYRF 22703
†Verimag is a joint laboratory of Université Joseph Fourier (Grenoble I), CNRS and INPG

an orthogonal point of view, since a mode structure is rather *sequential* than concurrent. This is typically the case with the modes of an aircraft, which can be as high level as "landing" mode, "take-off" mode, etc. The normal behavior of the system is a sequence of modes. In a transition between modes, the source mode is designed to build and guarantee a given configuration of the parameters of the system, such that the target mode can be entered.

1.1 Basic Lustre and its Standard Implementation

In a dataflow language for reactive systems, both the inputs and outputs of the system are described by their *flows* of values along time. Time is discrete and instants may be numbered by integers. If `x` is a flow, we will note x_n its value at the nth reaction (or nth *instant*) of the program.

A program consumes *input* flows and computes *output* flows, possibly using *local* flows which are not visible from the environment. Local and output flows are defined by *equations*. An equation "$\mathtt{x} = \mathtt{y} + \mathtt{z}$" defines the flow `x` from the flows `y` and `z` in such a way that, at each instant n, $x_n = y_n + z_n$.

A set of such equations, using arithmetic, Boolean, etc. operators, describes a network of operators, and is essentially equivalent to the description of a combinational circuit. The same constraints apply: one should not write sets of equations with instantaneous loops, like : $\{x = y + z, z = x + 1, ...\}$. This is a set of fixpoint equations that perhaps has solutions, but it is not accepted as a dataflow program.

In order to obtain the counterpart of *sequential* circuits, needed for the description of most reactive systems, one must reference the *past*. A single operator `pre` is introduced : $\forall n > 0, (\mathtt{pre}X)_n = X_{n-1}$.

One typically writes `T = pre(T) + i ;` , where `T` is an output, and `i` is an input. It means that, at each instant, the value of the flow `T` is obtained by adding the value of the current input `i` to the `pre`vious value of `T`. Initialization of flows is provided by the `->` operator. The equation `X = 0 -> pre(X) + 1` defines the flow of integers; as a reactive program, it produces values on the *basic clock*.

The straightforward implementation of a dataflow program in a sequential language is an infinite loop : first read inputs from the external world (read values from sensor devices for instance), then compute the corresponding outputs, depending on the current input and the history of inputs, emit the outputs (write values to actuator devices), and do so forever. Each pass in the loop corresponds to an instant of the basic clock.

The idea of the compilation is that, for each expression used in a `pre` operator, there is a variable in the sequential program. For instance `X = 0 -> pre(X) + 1 ; Y = i + X ;` is translated into:

```
int pX, X, i, Y ;  int init = 1 ;
while (1) {
     /* get inputs (call to external functions) : gives value to i*/
     /* The so-called ''reactive kernel'' of the system : */
     if (init) { X = 0      ; Y = i+X ; init = 0 ; }
     else      { X = pX + 1 ; Y = i+X ; }
     pX = X ;
     /* emit outputs (call to external functions) : writes X, Y*/
}
```

Optimizations in the compiler allow to write `Y = i+X ;` outside the conditional structure.

1.2 Talking about Running Modes in Lustre

In a dataflow language like Lustre, each output has a definition, which describes how it is updated, depending on its previous value, and on the values of other variables and inputs. The notion of *running mode* corresponds to the fact that there may exist several definitions (equations) for the same output, that should be used in distinct periods of time. For instance, the altitude of the aircraft is increasing in take-off mode, and decreasing in landing mode; the coordinates of a robot arm are computed with some equations when it moves right, and as soon as it reaches an obstacle, it begins moving left and the equations of the coordinates are entirely different.

Designing a system that clearly exhibits such "independent" *running modes* is not difficult since the mode structure can be encoded explicitly with the available dataflow constructs. Typically, some Boolean variables (or flows) *mode1, mode2* are used to identify the current mode, and all other variables computed by the system may have definitions of the form: *if (mode1) then ... else if (mode2) then ...* .

However the mode structure of the system is no longer readable in the resulting program, and modifying it is error prone. It cannot be used to improve the quality of the generated sequential code, because it is hidden in the conditionals in such a way that the information about modes is difficult to retrieve by compilers. A typical knowledge that could be used for improving the generated code (by reusing variables) is that modes are pairwise exclusive. Yet, whatever the encoding of modes into Boolean flows be, this exclusivity is hidden in the conditionals, and would have to be "recomputed" by the compiler.

1.3 A Proposal: Mode-Automata

As far as the readability of the mode structure is concerned, the users of SCADE expressed their desire to use a stategraph together with dataflow equations: a state corresponds to a mode; transitions in the stategraph are labeled by the conditions for changing modes; each state is associated with a dataflow program that describes how the outpout flows are to be computed, *in this mode*. Typically, a state may be labeled by `X = pre(X) + 1` and another state by `X = pre(X) - 1`, meaning that the output `X` of the system should evolve in two different ways, depending on the current mode. The transitions between states (modes) may be labeled by conditions on `X`. This makes the mode-structure appear as a whole, somewhere in the program; it is readable, especially if one provides a graphical syntax for stategraphs (which is what the SCADE developers aim at, since they already have a graphical and syntactic editor for dataflow networks). One may, for instance, modify the transitions between modes without modifying the dataflow programs attached to the modes.

Starting from these ideas, we proposed the mathematical model of *mode-automata* [MR98], which allows to mix dataflow programs used for the description of steady behaviors of systems, with automata traditionally used for describing sudden changes, interrupts, etc. Defining the semantics of mode-automata amounts to deciding precisely how to connect several dataflow programs sequentially, thus constructing the global behavior of a system from the behaviors of its various *modes*. Mode-automata can be viewed as a discrete form of *hybrid automata* [ACHH93, ACH$^+$95], where states are labeled by differential equations describing the behavior of a continuous environment; hybrid automata are used essentially for modelling purposes, in order to prove some properties of a discrete controller embedded in a physical environment. Mode-automata are used for *programming* purposes. The Lustre equations attached to states may have been (and generally are) obtained by discretizing differential equations.

We studied some properties related to mode-automata: trace semantics, equivalences, compositions, congruences, translation into pure Lustre, etc. The translation into pure Lustre is a way of defining the semantics of Mode-automata, but the Lustre program we obtain is essentially the one we would have written by hand: the mode structure is translated into conditionals. We have achieved the first goal, i.e., to propose a construct for talking about modes; but we did nothing at all for taking advantage of this new program structure (improving the generated code, transmitting hints to verification tools, ...) in the compiling process.

1.4 Related Work

We searched the litterature for a notion of *mode*, from a language point of view. We found very few relevant notions, all of them in domain-specific languages for real-time or reactive systems.

Modecharts [JM89] have recently joined the synchronous community, and we can find an Esterel-like semantics in [PSM96]. In this paper, modes are simply hierarchical and concurrent states like in Argos [Mar92]. In Modecharts, *modes* seem to be similar to the notion of task in Esterel, but they are not dealt with in the language itself; the mode structure of a reactive system may be described in Modecharts, but the behavior of the system in a particular mode is left to some host language that has no formal semantics.

In Statemate, one could probably describe a system with modes using a combination of statecharts [Har87] and activity charts, but there is no special construct for that.

The notion of mode closest to ours seems to be that of *Real-time mode-machines* described in [Pay96]. In this paper, modes are defined as collections of elementary states, and are dealt with in a specific way. However modes may not be exclusive, which differs from our model where modes are associated with the states of an automaton, hence pairwise exclusive.

On the other hand, there have been several attempts at adding some imperative features to dataflow synchronous languages like Lustre and Signal (or, symmetrically, some dataflow aspects to imperative synchronous languages like Esterel); see, among others, [RM95, CP95, JLMR94, And96]. Some of these approaches may be used for the description of *running modes*, in the sense adopted here, but none of them is specifically designed for that purpose; consequently, they do not propose a specific construct for the description of modes, and the mode structure of a system is still encoded somewhere. Nevertheless, we have been inspired by all these ideas.

1.5 Overview of the Paper

We first recall briefly how to extend the domain-specific dataflow language Lustre with a new construct that supports modes (see [MR98] for details, but the rest of the paper is understandable without them). Then this paper makes two contributions:

- We define the semantics of the extension in terms of *activation conditions* which is a way of generating code that knows about the mode-structure of the program. Activation conditions are defined in the intermediate format DC, used in the compilers of several synchronous languages (Lustre, Esterel, Signal, Argos...), and semantically close to the SCADE internal formats. The proof that this compilation scheme is consistent with the formal trace semantics of Mode-Automata is under development using PVS [ORS92].

- We propose a compilation method that reuses the official Lustre to DC compiler without modifying it ; this provides reasonable cost prototyping of language extensions. The translations into Lustre and into DC are both implemented.

2 Mode-Automata and their Translation into Pure Lustre

2.1 Mode-Automata: a Small Example

The mode-automaton of figure 1 describes a program that outputs an integer X. The initial value is 0. Then, the program has two *modes*: an incrementing mode, and a decrementing one. Changing modes is done according to the value reached by variable X: when it reaches 10, the mode is switched to "decrementing"; when X reaches 0 again, the mode is switched to "incrementing".

For simplicity, we give the definition for a simple case where the equations define only *integer* variables. One could easily extend this framework to all types of variables.

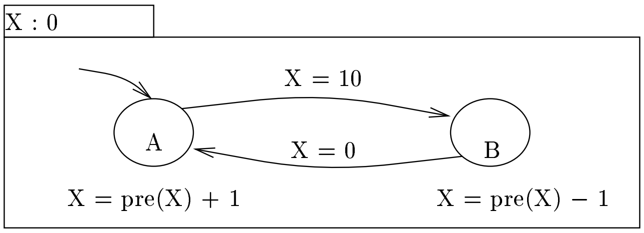

Figure 1: Mode-automata: a simple example

The translation into Lustre is made of two parts: the Boolean encoding of the automaton states and transitions (this provides one Boolean flow per mode); a sub network that describes how a given output X is obtained by connecting the outputs of all programs computing X through a multiplexor that selects the appropriate one depending on the current mode (given by the value of a Boolean flow computed in the first part).

We obtain (`->` is the initialization operator) :

```
A = true  -> (pre(A) and not (X=10)) or (pre(B) and (X=0))
B = false -> (pre(B) and not (X=0))  or (pre(A) and (X=10))
X = 0 -> if pre(A) then pre(X) + 1 else pre(X) - 1
```

In a dataflow language, this gives two sub-networks computing `pre(X) + 1` and `pre(X) - 1`. `X` is obtained by selecting the output of the appropriate one, depending on the current state. Without further analysis, the implementation into sequential code yields a program in which all the equations attached to modes are computed, at each instant. This raises two problems: efficiency, and side-effects.

First, recall that, in a dataflow language, conditionals are *strict*, which means that all branches are computed. In Lustre, writing "`Z = if (Y ≠ 0) then U/Y else U ;`" does not avoid the side-effect of the division by 0 (one has to use *clocks* — see below — in order to avoid this problem). In the context of mode-automata, one could write `Z = U/Y ; Y = 4` in one mode, and `Z = U` in another mode; the straightforward translation into Lustre, as described above, yields a program in which `U/Y` is computed at each instant, and may cause a division by 0; but the programmer perfectly knows that `U/Y` is needed only in a mode where `Y` is 4!

Second, when writing a mode-automaton, one clearly means that states are mutually exclusive, and reasonably hopes that, at a given instant in time, a single mode is active: the potentially complex equations attached to inactive modes should not be computed, and this could help saving time.

In order to obtain a more efficient and side-effect free Lustre translation, we need to specify that only one of the programs computing a given output X should be *activated* at each instant, since the system may not be in two distinct modes at the same time. This led us to a first conclusion: *modes* should act as *clocks*, not as *conditionals*.

2.2 Lustre Clocks and the Implementation of Mode-Automata

Consider a complex dataflow network, with two outputs X and Y. Assume the output X is used each second in the external world (meaning that the reactive system should have assigned a new value to it, each second) ; on the other hand, Y is needed only each hour. The straightforward implementation of the network will be a big loop, with one assignment per node of the network, and it should run sufficiently fast so that the X output is actually produced each second. A lot of time may be lost computing the values of some wires that are needed only for Y.

Clocks, as a language feature, are a tool that can be used by programmers in order to specify that some flows may be of no interest at some instants. The idea is to attach clocks to flows (i.e., wires of the network), not to nodes. In a network, operations may be applied only if all the input wires correspond to flows having the same clock. Specific operations for sampling and oversampling are used in order to connect flows that have distinct clocks.

When using clocks, one assumes that the information provided will actually be used by the compiler, at least for avoiding the computing of unecessary flows. Clocks are a real language feature, in the sense that they integrate perfectly in the semantics of Lustre. By the way, determining whether a network has correct clocks is a kind of type analysis [Cas92].

It seems that a mode-automaton should be translated into pure Lustre with clocks. This is a complex task. In some sense, this is like decompiling the quite imperative knowledge provided by the automaton into a declarative expression with clocks... that will have to be translated back to some imperative sequential language. Keeping track of the relevant information about exclusivity of states along this path seems hard and unecessarily complex.

3 Implementing Mode-Automata on top of the Lustre-to-DC Compiler

The other solution is to translate mode-automata directly into DC, which provides the more imperative notion of *activation condition*.

3.1 The Intermediate Format DC

DC, meaning "declarative code", is an intermediate code for the compilation of several synchronous languages into C. It is used as an intermediate form in the compilers of Lustre, Esterel, Signal, Argos. Having defined such an intermediate format allows the sharing of back-end tools like the C-code generator, the link editor, the simulator and the debugger, etc.

DC, like Lustre, has a declarative style. The main differences between Lustre and DC are the following: first, Lustre allows expressions of the form `pre(pre(pre... (X)))...)`, and the compiler is responsible for computing the minimal memory needed by this program; in DC, there is no `pre` construct, and the elements of memory are precisely identified. Second, while Lustre allows clocks attached to flows, DC provides a more imperative mechanism called *activation condition*.

Local and output flows can be defined by *definitions*, with which the *activation conditions* are associated. An activation condition is a Boolean flow. The meaning is that the defined flow is updated (according to its definition in terms of other flows) only when the activation condition has the value *true*, otherwise it keeps its previous value.We consider two kinds of definitions: equations and memorizations.

Equation defining X :	Memorization defining Y :
`X (init i) equcase`	`Y (init i) memocase`
`e1 at a1`	`e1 at a1`
`e2 at a2`	`e2 at a2`
`...`	`...`
`ek at ak`	`ek at ak`

These two constructs define the flows X and Y, both initialized with value i and computed, at each instant, depending on the value of the activation conditions `a1`, ... `ak`. The evaluation of activation conditions is sequential.

For equations we have:

$$X_0 = \begin{cases} e1_0 \text{ if } a1_0 \\ e2_0 \text{ if } \neg a1_0 \wedge a2_0 \\ ... \\ ek_0 \text{ if } \neg(a1_0 \vee ...) \wedge ak_0 \\ i \text{ if } \neg(a1_0 \vee ... \vee ak_0) \end{cases}, \qquad X_n = \begin{cases} e1_n \text{ if } a1_n \\ e2_n \text{ if } \neg a1_n \wedge a2_n \\ ... \\ ek_n \text{ if } \neg(a1_n \vee ...) \wedge ak_n \\ X_{n-1} \text{ if } \neg(a1_n \vee ... \vee ak_n) \end{cases} \quad \text{when } n > 0$$

For memorizations we have:

$$Y_0 = i, \qquad Y_{n+1} = \begin{cases} e1_n \text{ if } a1_n \\ e2_n \text{ if } \neg a1_n \wedge a2_n \\ ... \\ ek_n \text{ if } \neg(a1_n \vee ...) \wedge ak_n \\ Y_n \text{ if } \neg(a1_n \vee ... \vee ak_n) \end{cases}$$

3.2 Example

The DC program obtained for the mode-automaton of paragraph 2.1 above is the following:

```
inputs:    H bool ;  -- the basic clock
outputs:   X int ;
locals:
   M: int ;              -- the memory needed for X
   first: bool ;         -- true only at the first instant
   tBA, tAB : bool ;     -- transitions: from B to A, from A to B
   A, B : bool ;         -- states
definitions:
```

```
M                  memocase   :  X        at: true;
tBA                equcase    :  X = 0    at: B;
tAB                equcase    :  X = 10   at: A;
first (init true)  memocase   :  false    at: true;
B     (init false) memocase   :  tAB      at: A;      not tBA    at: B;
A     (init true)  memocase   :  not tAB  at: A;      tBA        at: B;
X                  equcase    :  0        at: first;  M + 1      at: A;    M - 1 at: B;
```

This gives input/output traces consistent with the semantics of mode-automata, with the additional properties that the equations attached to a mode are computed only when the system is in this mode. The definition of `X` is of particular interest: `M+1` (resp. `M - 1`) is computed only when the system is in state `A` (resp. `B`), from the same memory `M`.

The C program obtained from this DC program contains:

```
/* global variables */
      _boolean A, B, first ; int M, X ;
/* reset procedure */
      void reset () { first = true ; B = false ; A = true ; }
/* single-step procedure */
      void step () {
      /* buffers */
         _boolean __t1;
         _boolean __t2;
      /* equcase */
         if      (first) { X = 0 ; }
         else if (A) { X = M + 1 ; }
         else if (B) { X = M - 1 ; }
         if (B)      { tBA = (X==0) ; }
         if (A)      { tAB = (X==10) ; }
      /* memocase */
         M = X ; first = false ;
         if (A) { __t1 =  tAB ; }  else if (B) { __t1 = !tBA ; }
         if (A) { __t2 = !tAB ; }  else if (B) { __t2 =  tBA ; }
         B = __t1; A = __t2;
}
```

Now, recall that `M+1` could be replaced by a complex program describing the behavior of the system in mode `A`. The C code that computes this expression and assigns it to variable `X` is indeed embedded into a `if (A)`.

3.3 Correctness of the Compilation Scheme

The compilation scheme used for translating mode-automata into DC heavily uses the activation condition mechanism, and it is not obvious that the DC program behaves as expected.

The correctness of the compilation scheme can be formalized as follows: both mode-automata and DC have a formal trace semantics, i.e. there exists a mathematically defined function f_m from mode-automata to traces, and another function f_d from DC programs to traces. Traces are sets of sequences of input/output tuples. The compilation scheme can be viewed as a function Y from mode-automata to DC. We have to prove that :

$$\forall M. f_m(M) = f_d(Y(M))$$

This proof could be performed by hand. However, dealing with the complex definitions of mode-automata and DC is likely to be error prone in such a proof. We chose to use the tool PVS [ORS92] in order to mechanize the proof. This work is under development and, after the necessary amount of time spent learning PVS, we encountered no difficulty. The idea is to define mode-automata, DC programs and traces as abstract datatypes in PVS, and the three translation functions f_m, f_d and Y as recursive functions from one datatype to another. Proving the equality of traces amounts to an induction on the indices of the sequences, which model time.

3.4 Implementation

In the paragraph above, we only discussed the compilation scheme, i.e. the mathematically defined function from mode-automata to DC. We are now concerned with the actual implementation of this function. The main point is that mode-automata have parts written in Lustre, for which we have to produce DC code, and we should not rewrite a Lustre-to-DC compiler. Consequently, a mode-automaton is first split into two parts: the automaton, and the Lustre parts (equations attached to states, conditions labeling the transitions). The automaton is encoded into DC, directly. The Lustre parts are first encapsulated into syntactically correct Lustre programs, then passed to the official Lustre-to-DC compiler. The DC code obtained is analysed, and relevant code is extracted and merged into the automaton encoding. This gives the DC code for the mode-automaton.

The only constraint we put on the Lustre-to-DC compiler is that it should be usable with no optimizations: it should not re-use variables in the DC code, or perform dead-code elimination. This is not too strong a constraint, since we have the same for the Lustre debugger.

4 Conclusions and Future Work

Starting from a demand from the users of the industrial programming environment SCADE, we defined *mode-automata*, and studied their trace semantics and mathematical properties. Then we used this mathematical model to define an extension of Lustre. We proposed a compilation scheme that uses the official Lustre compiler without modifying it. This is implemented. The proof of this compilation scheme is under development using PVS.

4.1 Evaluation of the approach

The prototype implementation has been used for two case studies: a simplified temperature control system for an aircraft, submitted by SAAB and already studied in the SYRF[1] project; an operational nuclear plant controller submitted by Schneider Electric. The latter was already written in pure Lustre, and we performed a kind of reverse engineering task, in order to identify *modes* in the behavior of the system; we ended up with a mode-automata program, and we translated it into pure Lustre automatically. We were able to formally and automatically *prove* that the two programs were trace equivalent, using the model-checker Lesar [HLR92].

Further work on introducing modes in Lustre includes some easy extensions (variables local to a state and its outgoing transitions; priorities between transitions sourced in the same state, ...) and some extensions of the subset of Lustre we allow to label states (calling nodes with memory, using Lustre

[1]SYRF means "Synchronous Reactive Formalisms". SYRF is Esprit "Long Term Research" Project 22703

clocks, ...) that require a careful study of the interaction between the automaton structure and the Lustre constructs. We shall also allow imported objects from a host language, like in Lustre.

The fundamental extension we have in mind is to allow compositions of mode-automata. We already defined the parallel composition in [MR98]; it should be easy to implement, since it corresponds perfectly to the intrinsic dataflow parallelism of Lustre or DC nodes. The other composition is the *hierarchy* of modes, since a system described using modes is likely to use also *sub-modes*. Here the main question is: what information should be shared by the sub-modes of a given mode M? More accurately: should the sub-modes of M share the equations defining some variables, or should they share transitions, or both? If they share transitions, we can reuse the work on the refinement operation of Argos [Mar92], were a transition leaving a state is indeed applicable to all the sub-states (Argos was inspired by Statecharts [Har87], and the refinement operation corresponds to the notion of OR-state).

Note that the work developed for proving the compilation scheme using PVS can be reused when defining compositions of mode-automata, in order to prove consistency between the trace semantics of the compositions and the behavior of the DC programs we propose to implement them.

Finally, we already said that DC is close to SCADE (the idea of activation conditions orginally came from the designers of SCADE, indeed). Hence the translation of mode-automata into one of the SCADE internal formats should be straightforward, as far as the semantics is concerned; moreover, the compilation scheme is proved. However, new problems may appear, concerning the integration of a tool for editing modes, in the SCADE programming environment, which already proposes a syntactic and graphical editor for dataflow networks, and guarantees the synchronization between several views of the same program.

4.2 Generalization of the approach

The work described here was motivated by a very particular problem, and we proposed first a particular solution. However, we now see two directions for generalization: first, the whole approach — definition of a new construct, semantics in DC, influence on the generated code — should be applicable to other extensions of Lustre, with reasonable prototyping cost; this is interesting for answering the demands of the users.

Second — and this is a direction for more academic research, at least at the beginning — we would like to explore the influence of the mode-structure of a system on the structuring of its proof. In the field of formal languages for reactive and real-time systems, the *assume-guarantee* rule is well known: it allows the compositional proof of a *parallel* system. The idea is to prove each component under some assumptions on the behaviour of the other. We would like to find a method for proving a mode program, which would allow the decomposition of the global proof into smaller proofs, each of them being performed on one mode, with some assumptions on the others.

As we said above, we found very few references on the notion of mode in the litterature. However, it is still possible that the notion already exist, with another name. We would be pleased to hear about any relevant attempt at providing, in a programming language — be it domain specific or general purpose — a construct that let the programmer express the "mode" structure of the problem he has in mind, in such a way that this information be usable for automatic tools like the code generator or a proof tool.

4.3 The full paper

In the full paper we will report on the PVS proof, and give the formal definition of the translation from mode-automata to DC.

References

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science B*, 138:3–34, January 1995.

[ACHH93] R. Alur, C. Courcoubetis, T. A. Henzinger, and Pei-Hsin Ho. Hybrid automata: an algorithmic approach to the specification and analysis of hybrid systems. In *Workshop on Theory of Hybrid Systems*, Lyngby, Denmark, October 1993. LNCS 736, Springer Verlag.

[And96] C. André. Representation and analysis of reactive behaviors: a synchronous approach. In *IEEE-SMC'96, Computational Engineering in Systems Applications*, Lille, France, July 1996.

[BB91] A. Benveniste and G. Berry. Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9), September 1991.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[Cas92] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*, Munich, January 1987.

[CP95] P. Caspi and M. Pouzet. A functional extension to LUSTRE. In *Eighth International Symp. on Languages for Intensional Programming, ISLIP'95*, Sidney, May 1995.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.

[Har87] D. Harel. Statecharts : A visual approach to complex systems. *Science of Computer Programming*, 8:231–275, 1987.

[HCRP91] 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.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Ingeneering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[JLMR94] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond. A multiparadigm language for reactive systems. In *In 5th IEEE International Conference on Computer Languages*, Toulouse, May 1994. IEEE Computer Society Press.

[JM89] F. Jahanian and A. Mok. Modechart, a specification language for real time systems. *IEEE Transactions on Software Engineering*, 1989.

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*. LNCS 630, Springer Verlag, August 1992.

[MR98] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer Verlag, LNCS 1381.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: a prototype verification system. In *11th Conf. on Automated Deduction*. Lecture Notes in Computer Science 607, Springer Verlag, 1992.

[Pay96] S. Paynter. Real-time mode-machines. In *Formal Techniques for Real-Time and Fault Tolerance (FTRTFT)*, pages 90–109. LNCS 1135, Springer Verlag, 1996.

[PSM96] Carlos Puchol, Douglas Stuart, and Aloysius K. Mok. An operational semantics and a compiler for modechart specificiations. Technical Report CS-TR-95-37, University of Texas, Austin, July 1, 1996.

[RM95] E. Rutten and F. Martinez. SIGNALGTI, implementing task preemption and time interval in the synchronous data-flow language SIGNAL. In *7th Euromicro Workshop on Real Time Systems*, Odense (Denmark), June 1995.