Mode-Automata: About Modes and States for Reactive Systems*

Florence Maraninchi and Yann Rémond

VERIMAG** – Centre Equation, 2 Av. de Vignate – F38610 GIERES
http://www.imag.fr/VERIMAG/PEOPLE/Florence.Maraninchi

Abstract. In the field of reactive system programming, dataflow synchronous languages like Lustre [BCH$^+$85,CHPP87] or Signal [GBBG85] offer a syntax similar to block-diagrams, and can be efficiently compiled into C code, for instance. Designing a system that clearly exhibits several "independent" *running modes* is not difficult since the mode structure can be encoded explicitly with the available dataflow constructs. However the mode structure is no longer readable in the resulting program; modifying it is error prone, and it cannot be used to improve the quality of the generated code.
We propose to introduce a special construct devoted to the expression of a mode structure in a reactive system. We call it *mode-automaton*, for it is basically an automaton whose states are labeled by dataflow programs. We also propose a set of operations that allow the composition of several mode-automata (parallel and hierarchic compositions taken from Argos [Mar92]), and we study the properties of our model, like the existence of a congruence of mode-automata for instance, as well as implementation issues.

1 Introduction

The work on which we report here has been motivated by the need to talk about running modes in a dataflow synchronous language.

Dataflow languages like Lustre [BCH$^+$85,CHPP87] or Signal [GBBG85] belong to the family of *synchronous* languages [BB91] devoted to the design, programming and validation of reactive systems. They have a formal semantics and can be efficiently compiled into C code, for instance.

The dataflow style is clearly appropriate when the behaviour of the system to be described has some regularity, like in signal-processing. Designing a system that clearly exhibits several "independent" *running modes* is not so difficult since the mode structure can be encoded explicitly with the available dataflow constructs. However the mode structure is no longer readable in the resulting program; modifying it is error prone, and it cannot be used to improve the

* 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 quality of the generated code, decompose the proofs or at least serve as a guide in the analysis of the program.

In section 2 we propose a definition of a mode, in order to motivate our approach. Section 3 defines *mini-Lustre*, a small subset of Lustre which is sufficient for presenting our notion of mode-automata in section 4. Section 5 explains how to compose these mode-automata, with the operators of Argos, and Section 6 defines a congruence. Section 7 compares the approach to others, in which modes have been studied. Finally, section 8 gives some ideas for further work.

2 What is a Mode?

One (and perhaps the only one) way of facing the complexity of a system is to decompose it into several "independent" tasks. Of course the tasks are never completely independent, but it should be possible to find a decomposition in which the tasks are not too strongly connected with each other — i.e. in which the interface between tasks is relatively simple, compared to their internal structure. The tasks correspond to some abstractions of the global behaviour of the system, and they may be viewed as differents parts of this global behaviour, 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 an orthogonal point of view, since a mode structure is rather *sequential* than concurrent. This is typically the case with the modes of an airplane, which can be as high level as "landing" mode, "take-off" mode, etc. The normal behaviour 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. On the other hand, modes may be divided into sub-modes.

Of course the mode structure may interfere with the concurrent structure, in which case each concurrent subsystem may have its own modes, and the global view of the system shows a Cartesian product of the sets of modes. Or the main view of the system may be a set of modes, and the description of each mode is further decomposed into concurrent tasks. Hence we need a richer notion of mode.

This seems to give something similar to the notion of mode we find in Modecharts [JM88], where modes are structured like in Statecharts [Har87] (an And/Or tree). However, see section 7 for more comments about Modecharts, and a comparison between Modecharts and our approach.

2.1 Modes and States

Technically, all systems can be viewed as a (possibly huge, or even infinite) set of elementary and completely detailed states, such that the knowledge about the current state is sufficient to determine the correct output, at any point in time. States are connected by *transitions*, whose firing depends on the inputs to the system. This complete model of the system behaviour may not be manageable, but it exists. We call its states and transitions *execution* states and transitions. Execution states are really concrete ones, related for instance to the content of the program memory during execution.

The question is: how can we define the modes of a system in terms of its execution states and transitions?

Since the state-transition view of a complex behaviour is intrinsically sequential, it seems that, in all cases, it should be possible to relate the abstract notion of mode to *collections* of execution states. The portion of behaviour corresponding to a mode is then defined as a set of execution states together with the attached transitions. Related questions are: are these collections disjoint? do they cover the whole set of states? S. Paynter [Pay96] suggests that these two questions are orthogonal, and defines *Real-Time Mode-Machines*, which describe exhaustive but not necessarily exclusive modes (see more comments on this paper in section 7). In fact, the only relevant question is that of exclusivity, since, for non exhaustive modes, one can always consider that the "missing" states form an additional mode.

2.2 Talking about Modes in a Programming Language

All the formalisms or languages defined for reactive systems offer a parallel composition, together with some synchronization and communication mechanism. This operation supports a conceptual decomposition in terms of concurrent tasks, and the parallel structure can be used for compositional proofs, generation of distributed code, etc.

The picture is not so clear for the decomposition into modes. The question here is how to use the mode structure of a complex system for programming it, i.e. what construct should we introduce in a language to express this view of the system? The mode structure should be as readable in a program as the concurrent structure is, thus making modifications easier; moreover, it should be usable to improve the quality of the generated code, or to serve as a guide for decomposing proofs.

The key point is that *it should be possible to project a program onto a given mode, and obtain the behaviour restricted to this mode* (as it is usually possible to project a parallel program onto one of its concurrent components, and get the behaviour restricted to this component).

2.3 Modes and Synchronous Languages

None of the existing synchronous languages can be considered as providing a construct for expressing the mode structure of a reactive system.

We are particularly interested in dataflow languages. When trying to think in terms of modes in a dataflow language, one has to face two problems: first, there should be a way to express that some parts of a program are not always active (and this is not easy); second, if these parts of the program represent different modes, there should be a way of describing how the modes are organized into the global behaviour of the system.

Several proposals have been made for introducing some *control* features in a dataflow program, and this has been tried for one language or another among the synchronous family: [RM95] to introduce in Signal a way to define *intervals* delimited by some properties of the inputs, and to which the activity of some subprograms can be attached; [JLMR94,MH96] propose to mix the automaton constructs of Argos with the dataflow style of Lustre: the refinement operation of Argos allows to refine a state of an automaton by a (possibly complex) subsystem. Hence the activity of subprograms is attached to states. Embedding Lustre nodes in an Esterel program is possible, and would have the same effect.

However, providing a full set of start-and-stop control structures for a dataflow language does not necessarily improve the way modes can be dealt with. It solves the first problem mentioned above, i.e. the control structures taken in the imperative style allow the specification of activity periods of some subprograms described in a dataflow declarative style. But it does little for the second problem: a control structure like the interrupt makes it easy to express that the system switches between different behaviours, losing information about the current state of the behaviour that is interrupted, and starting a new one in some initial configuration. Of course, some information may be transmitted from the behaviour that is killed to the one that is started, but this is not the default, and it has to be expressed explicitly, with the communication mechanism for instance. For switching between *modes*, we claim that the emphasis should be on what is *transmitted* from one mode to another. Transmitting the whole configuration reached by the system should be the default if we consider that 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.

2.4 A Proposal: Mode-Automata

We propose a programming model called "*mode-automata*", made of: operations on automata taken from the definition of Argos [Mar92]; dataflow equations taken from Lustre [BCH+85]. We shall see that mode-automata can be considered as a discrete version of hybrid automata [MMP91], in which the states are labeled by systems of differential equations that describe how the continuous environment evolves. In our model, states represent the running *modes* of a system, and the equations associated with the states could be obtained by discretizing the control laws. Mode-automata have the property that a program may be projected onto one of its modes.

3 Mini-Lustre: a (very) Small Subset of Lustre

For the rest of the paper, we use a very small subset of Lustre. A program is a single node, and we avoid the complexity related to types as much as possible. In some sense, the mini-Lustre model we present below is closer to the DC [CS95] format used as an intermediate form in the Lustre, Esterel and Argos compilers.

Definition 1 (mini-Lustre programs). *$N = (V_i, V_o, V_l, f, I)$ where: V_i, V_o and V_l are pairwise disjoint sets of* input, output *and* local *variable names. I is a total function from $V_o \cup V_l$ to constants. f is a total function from $V_o \cup V_l$ to the set $Eq(V_i \cup V_o \cup V_l)$ and $Eq(V)$ is the set of expressions with variables in V, defined by the following grammar: $e ::= c \mid x \mid op(e, ..., e) \mid pre(x)$. c stands for constants, x stands for a name in V, and op stands for all combinational operators. An interesting one is the conditional* `if` e_1 `then` e_2 `else` e_3 *where* e_1 *should be a Boolean expression, and* e_2, e_3 *should have the same type. $pre(x)$ stands for the* previous *value of the flow denoted by x. In case one needs $pre(x)$ at the first instant, $I(x)$ should be used.* □

We restrict mini-Lustre to integer and Boolean values. All expressions are assumed to be typed correctly. As in Lustre, we require that the dependency graph between variables be acyclic. A dependency of X onto Y appears whenever there exists an equation of the form $X = ...Y...$ and Y does not appear inside a `pre` operator. In the syntax of mini-Lustre programs, it means that Y appears in the expression $f(X)$, not in a `pre` operator.

Definition 2 (Trace semantics of mini-Lustre). *Each variable name v in the mini-Lustre program describes a* flow *of values of its type, i.e. an infinite sequence $v_0, v_1, ...$. Given a sequence of inputs, i.e. the values v_n, for each $v \in V_i$ and each $n \geq 0$, we describe below how to compute the sequences (or* traces*) of local and output flows of the program. The initialization function gives values to variables for the instant "before time starts", since it provides values in case* `pre`(x) *is needed at instant 0. Hence we can call it x_{-1}:*

$$\forall v \in V_o \cup V_l. \quad v_{-1} = I(v)$$

For all instants in time, the value of an output or local variable is computed according to its definition as given by f:

$$\forall n \geq 0. \quad \forall v \in V_o \cup V_l. \quad v_n = f(v)[x_n/x][x_{n-1}/pre(x)]$$

We take the expression $f(v)$, in which we replace each variable name x by its current value x_n, and each occurrence of `pre`(x) *by the previous value x_{n-1}. This yields an expression in which combinational operators are applied to constants. The set of equations we obtain for defining the values of all the flows over time is acyclic, and is a sound definition.* □

Definition 3 (Union of mini-Lustre nodes). *Provided they do not define the same outputs, i.e.* $\mathcal{V}_o^1 \cap \mathcal{V}_o^2 = \emptyset$, *we can put together two mini-Lustre programs. This operation consists in connecting the outputs of one of them to the inputs of the other, if they have the same name. These connecting variables should be removed from the inputs of the global program, since we now provide definitions for them. This corresponds to the usual dataflow connection of two nodes.*

$$(\mathcal{V}_i^1, \mathcal{V}_o^1, \mathcal{V}_l^1, f^1, I^1) \cup (\mathcal{V}_i^2, \mathcal{V}_o^2, \mathcal{V}_l^2, f^2, I^2) =$$
$$((\mathcal{V}_i^1 \cup \mathcal{V}_i^2) \setminus \mathcal{V}_o^1 \setminus \mathcal{V}_o^2, \quad \mathcal{V}_o^1 \cup \mathcal{V}_o^2, \quad \mathcal{V}_l^1 \cup \mathcal{V}_l^2,$$
$$\lambda x. \mathit{if}\ x \in \mathcal{V}_o^1 \cup \mathcal{V}_l^1\ \mathit{then}\ f^1(x)\ \mathit{else}\ f^2(x),$$
$$\lambda x. \mathit{if}\ x \in \mathcal{V}_o^1\ \mathit{then}\ I^1(x)\ \mathit{else}\ I^2(x))$$

Local variables should be disjoint also, but we can assume that a renaming is performed before two mini-Lustre programs are put together. Hence $\mathcal{V}_l^1 \cap \mathcal{V}_l^2 = \emptyset$ *is guaranteed. The union of sets of equations should still satisfy the acyclicity constraint.* □

Definition 4 (Trace equivalence for mini-Lustre). *Two programs* $L_1 = (\mathcal{V}_i, \mathcal{V}_o, \mathcal{V}_l^1, f^1, I^1)$ *and* $L_2 = (\mathcal{V}_i, \mathcal{V}_o, \mathcal{V}_l^2, f^2, I^2)$ *having the same input/output interface are trace-equivalent (denoted by* $L_1 \sim L_2$*) if and only if they give the same sequence of outputs when fed with the same sequence of inputs.* □

Definition 5 (Trace equivalence for mini-Lustre with no initial specification). *We consider mini-Lustre programs without initial specification, i.e. mini-Lustre programs without the function I that gives values for the flows "before time starts". Two such objects* $L_1 = (\mathcal{V}_i, \mathcal{V}_o, \mathcal{V}_l^1, f^1)$ *and* $L_2 = (\mathcal{V}_i, \mathcal{V}_o, \mathcal{V}_l^2, f^2)$ *having the same input/output interface are trace-equivalent (denoted by* $L_1 \approx L_2$*) if and only if,* for all initial configuration *I, they give the same sequence of outputs when fed with the same sequence of inputs.* □

Property 1 : Trace equivalence is preserved by union
$L \sim L' \Longrightarrow L \cup M \sim L' \cup M$ □

4 Mode-Automata

4.1 Example and Definition

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.

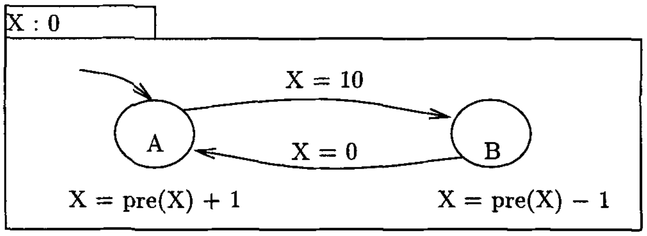

Fig. 1. Mode-automata: a simple example

Definition 6 (Mode-automata).
A mode-automaton is a tuple $(Q, q_0, \mathcal{V}_i, \mathcal{V}_o, \mathcal{I}, f, T)$ *where:*

- Q *is the set of* states *of the automaton part*
- $q_0 \in Q$ *is the* initial *state*
- $\mathcal{V}_i$ *and* $\mathcal{V}_o$ *are sets of names for* input *and* output *integer* variables.
- $T \subseteq Q \times C(\mathcal{V}) \times Q$ *is the set of* transitions, *labeled by* conditions *on the variables of* $\mathcal{V} = \mathcal{V}_i \cup \mathcal{V}_o$
- $\mathcal{I} : \mathcal{V}_o \longrightarrow Z$ *is a function defining the* initial *value of output variables*
- $f : Q \longrightarrow \mathcal{V}_o \longrightarrow \mathsf{EqR}$ *defines the labeling of states by total functions from* $\mathcal{V}_o$ *to the set* $\mathsf{EqR}(\mathcal{V}_i \cup \mathcal{V}_o)$ *of expressions that constitute the right parts of the* equations *defining the variables of* $\mathcal{V}_o$.

The expressions in $\mathsf{EqR}(\mathcal{V}_i \cup \mathcal{V}_o)$ *have the same syntax as in mini-Lustre nodes:* $e ::= c \mid x \mid op(e, ..., e) \mid \texttt{pre}(x)$, *where* c *stands for constants,* x *stands for a name in* $\mathcal{V}_i \cup \mathcal{V}_o$, *and* op *stands for all combinational operators. The conditions in* $C(\mathcal{V}_i \cup \mathcal{V}_o)$ *are expressions of the same form, but without* `pre` *operators; the type of an expression serving as a condition is Boolean.* □

Note that *Input* variables are used only in the right parts of the equations, or in the conditions. *Output* variables are used in the left parts of the equations, or in the conditions.
We require that the automaton part of a mode-automaton be *deterministic*, i.e., for each state $q \in Q$, if there exist two outgoing transitions (q, c_1, q_1) and (q, c_2, q_2), then $c_1 \wedge c_2$ is not satisfiable.
We also require that the automaton be *reactive*, i.e., for each state $q \in Q$, the formula $\bigvee_{(q,c,q') \in T} c$ is true.
With these definitions, the example of figure 1 is written as:

$$(\{A, B\}, A, \emptyset, \{X\}, I : X \to 0,$$
$$f(A) = \{ \texttt{X = pre(X) + 1} \}, f(B) = \{ \texttt{X = pre(X) - 1} \}),$$
$$\{(A, \texttt{X = 10}, B), (B, \texttt{X = 0}, A), (A, \texttt{X} \neq \texttt{10}, A), (B, \texttt{X} \neq \texttt{0}, B)\})$$

In the graphical notation of the example, we omitted the two loops $(A, \texttt{X} \neq \texttt{10}, A)$ and $(B, \texttt{X} \neq \texttt{0}, B)$.

4.2 Semantics by Translation into Mini-Lustre

The main idea is to translate the automaton structure of a mode-automaton into mini-Lustre, in a very classical and straightforward way. Then we gather all the sets of equations attached to states into a single conditional structure. We choose to encode each state by a Boolean variable. Arguments for a more efficient encoding exist, and such an encoding could be applied here, independently from the other part of the translation. However, it is sometimes desirable that the pure Lustre program obtained from mode-automata be *readable*; in this case, a clear (and one-to-one) relation between states in the mode-automaton, and variables in the Lustre program, is required.

The function $\mathcal{L}$ associates a mini-Lustre program with a mode-automaton. We associate a Boolean local variable with each state in $Q = \{q_0, q_1, ..., q_n\}$, with the same name. Hence:

$$\mathcal{L}((Q, q_0, \mathcal{V}_i, \mathcal{V}_o, \mathcal{I}, f, T)) = (\mathcal{V}_i, \mathcal{V}_o, Q, e, J)$$

The initial values of the variables in $\mathcal{V}_o$ are given by the initialization function $\mathcal{I}$ of the mode-automaton, hence $\forall x \in \mathcal{V}_o,\ J(x) = \mathcal{I}(x)$. For the local variables of the mini-Lustre program, which correspond to the states of the mode-automaton, we have: $J(q_0) = \text{true}$ and $J(q) = \text{false}, \forall q \neq q_0$.

The equation for a local variable q that encodes a state q expresses that we are in state q at a given instant if and only if we were in some state q', and a transition (q', c, q) could be taken. Note that, because the automaton is reactive, the system can always take a transition, in any state. A particular case is $q' = q$: staying in a state means taking a loop on that state, at each instant

$$\text{for } q \in Q, e(q) \text{ is the expression:} \quad \bigvee_{(q',c,q)\in T} \text{pre } (q' \wedge c)$$

For $x \in \mathcal{V}_o$, $e(x)$ is the expression :

$$\text{if } q_0 \text{ then } f(q_0) \text{else if } q_1 \text{ then } f(q_1) \text{...else if } q_n \text{ then } f(q_n)$$

The mini-Lustre program obtained for the example is the following (note that `pre(A and X = 10)` is the same as `pre(A) and pre(X) = 10`, hence the equations have the form required in the definition of mini-Lustre).

$\mathcal{V}_i = \emptyset \qquad \mathcal{V}_o = \{X\} \qquad \mathcal{V}_l = \{A, B\}$
$f(X)$: `if A then pre(X)+1 else pre(X)-1`
$f(A)$: `pre (A and not X=10) or pre(B and X = 0)`
$f(B)$: `pre (B and not X=0) or pre(A and X = 10)`
$I(X) = 0 \qquad I(A) = true \qquad I(B) = false$

5 Compositions of Mode-Automata

5.1 Parallel Composition with Shared Variables

A single mode-automaton is appropriate when the structure of the running modes is flat. Parallel composition of mode-automata is convenient whenever the modes can be split into at least two orthogonal sets, such that a set of modes is used for controlling some of the variables, and another set of modes is used for controlling other variables.

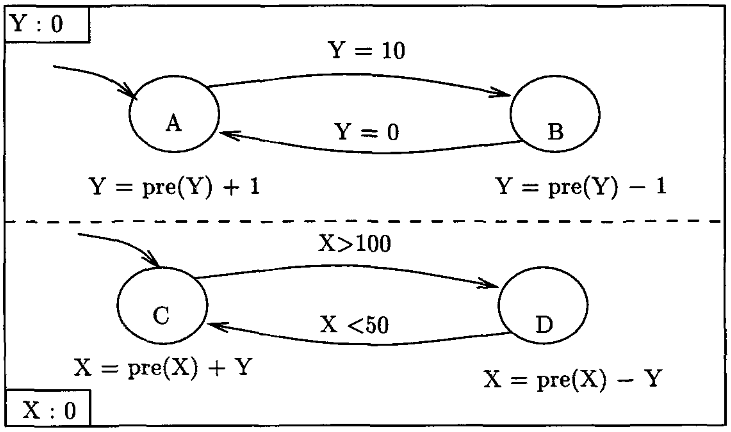

Fig. 2. Parallel composition of mode-automata, an example

Definition Provided $\mathcal{V}_o^1 \cap \mathcal{V}_o^2 = \emptyset$, we define the parallel composition of two mode-automata by :

$$(Q^1, q_0^1, T^1, \mathcal{V}_i^1, \mathcal{V}_o^1, \mathcal{I}^1, f^1) \| (Q^2, q_0^2, T^2, \mathcal{V}_i^2, \mathcal{V}_o^2, \mathcal{I}^2, f^2) = \\ (Q^1 \times Q^2, (q_0^1, q_0^2), (\mathcal{V}_i^1 \cup \mathcal{V}_i^2) \setminus \mathcal{V}_o^1 \setminus \mathcal{V}_o^2, \mathcal{V}_o^1 \cup \mathcal{V}_o^2, \mathcal{I}, f)$$

Where:

$$f(q^1, q^2)(X) = \begin{cases} f^1(q^1)(X) \text{ if } X \in \mathcal{V}_o^1 \\ f^2(q^2)(X) \text{ otherwise, i.e. if } X \in \mathcal{V}_o^2 \end{cases}$$

Similarly:

$$\mathcal{I}(X) = \begin{cases} \mathcal{I}^1(X) \text{ if } X \in \mathcal{V}_o^1 \\ \mathcal{I}^2(X) \text{ if } X \in \mathcal{V}_o^2 \end{cases}$$

And the set T of global transitions is defined by:

$$(q^1, C^1, q'^1) \in T^1 \quad \wedge (q^2, C^2, q'^2) \in T^2 \qquad \Longrightarrow \quad ((q^1, q^2), C^1 \wedge C^2, (q'^1, q'^2)) \in T$$

The following property establishes a relationship between the parallel composition defined for mode-automata as a synchronous product, and the intrinsic parallelism of Lustre programs, captured by the union of sets of equations:

Property 2
$\mathcal{L}(M_1 \| M_2) \sim \mathcal{L}(M_1) \cup \mathcal{L}(M_2)$ □

5.2 Ideas for Hierarchic Modes

A very common notion related to modes is that of ***sub-modes***. In the hierarchic composition below, the equation `X = pre(X) – 1` associated with state D is shared by the two submodes, while these submodes have been introduced for defining Y more precisely. Note that the scope of Y is the whole program, not only state D. Splitting state D for refining the definition of Y has something to do with equivalences of mode-automata (see below). Indeed, we would like the hierarchic composition to be defined in such a way that the program of figure 3 be "equivalent" to that of figure 4, where the program associated with state D uses a Boolean variable q. "Refining" (or exploding) the conditional definition of Y into the mode-automaton with states A, B of figure 3 is the work performed by the Lustre compiler, when building an interpreted automaton from a Lustre program.

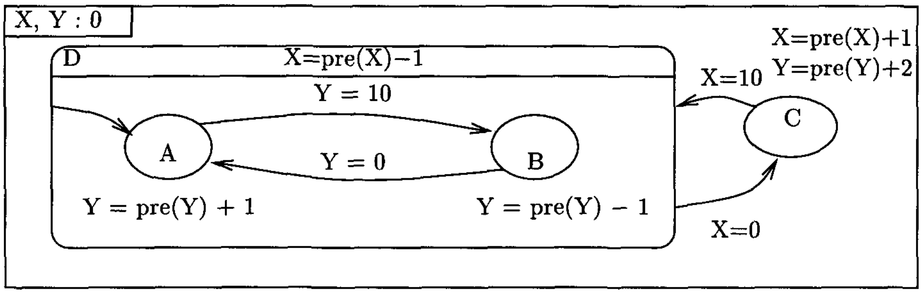

Fig. 3. Hierarchic composition of mode-automata, an example

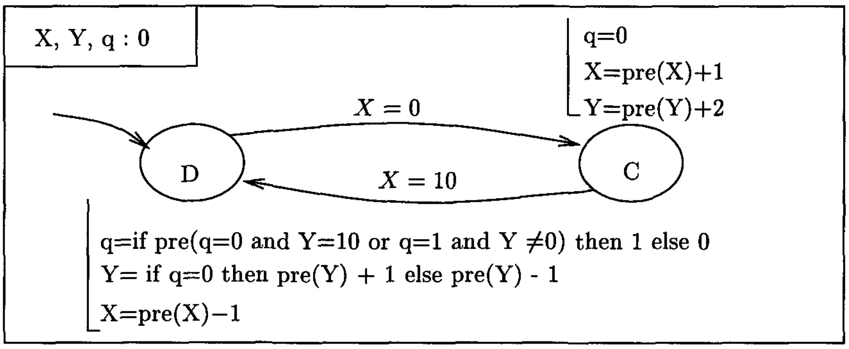
X, Y, q : 0
q=0
X=pre(X)+1
Y=pre(Y)+2
X = 0
D
X = 10
C
q=if pre(q=0 and Y=10 or q=1 and Y ≠0) then 1 else 0
Y= if q=0 then pre(Y) + 1 else pre(Y) - 1
X=pre(X)−1

Fig. 4. Describing Y with a conditional program ($q = 0$ corresponds to A; $q = 1$ corresponds to B)

6 Congruences of Mode-Automata

We try to define an equivalence relation for mode-automata, to be a congruence for the parallel composition.

There are essentially two ways of defining such an equivalence : either as a relation induced by the existing trace equivalence of Lustre programs; or by an explicit definition on the structure of mode-automata, inspired by the trace equivalence of automata. The idea is that, if two states have equivalent sets of equations, then they can be identified.

Definition 7 (Induced equivalence of mode-automata).
$M_1 \equiv_i M_2 \iff \mathcal{L}(M_1) \sim \mathcal{L}(M_2)$ *(see definition 4 for $\sim$).* □

Definition 8 (Direct equivalence of mode-automata). *The direct equivalence is a bisimulation, taking the labeling of states into account:*

$$(Q^1, q_0^1, T^1, \mathcal{V}_i^1, \mathcal{V}_o^1, \mathcal{I}^1, f^1) \equiv_d (Q^2, q_0^2, T^2, \mathcal{V}_i^2, \mathcal{V}_o^2, \mathcal{I}^2, f^2) \iff$$
$\exists R \subseteq R_s$ *such that:*
$$(q_0^1, q_0^2) \in R \quad \wedge$$
$$(q^1, q^2) \in R \implies \begin{cases} (q^1, c^1, q'^1) \in T^1 \implies & \exists q'^2, c^2 \text{ s. t. } (q^2, c^2, q'^2) \in T^2 \\ & \wedge\ (q'^1, q'^2) \in R \\ & \wedge\ c^1 = c^2 \\ \text{and conversely.} & \end{cases}$$

Where $R_s \subseteq Q^1 \times Q^2$ is the relation on states induced by the equivalence of the attached sets of equations: $(q^1, q^2) \in R_s \iff f^1(q^1) \approx f^2(q^2)$ (see definition 5 for $\approx$). □

Note: the conditions labeling transitions are Boolean expressions with subexpressions of the form $a\#b$, where a and b are integer constants or variables, and $\#$ is a comparison operator yielding a Boolean result. We can always consider that these conditions are expressed as sums of monomials, and that a transition labeled by a sum $c \vee c'$ is replaced by two transitions labeled by c and c' (between the same states). Hence, in the equivalence relation defined above, we can require that the conditions c^1 and c^2 match. Since $c^1 = c^2$ is usually undecidable (it is not the syntactic identity), the above definition is not practical.

It is important to note that the two equivalences *do not* coincide: $M_1 \equiv_d M_2 \implies M_1 \equiv_i M_2$, but this is not an equivalence. Translating the mode-automaton into Lustre before testing for equivalence provides a global comparison of two mode-automata. On the contrary, the second definition of equivalence compares subprograms attached to states, and may fail in recognizing that two mode-automata describe the same global behaviour, when there is no way of establishing a correspondence between their states (see example below).

Example 1. Let us consider a program that outputs an integer X. X has three different behaviours, described by: $B_1 : X = pre(X) + 1$, $B_2 : X = pre(X) + 2$

and $B_3 : X = pre(X) - 1$. The transitions between these behaviours are triggered by conditions on X: C_{ij} is the condition for switching from B_i to B_j.

A mode-automaton M_1 describes B_1 and B_2 with a state q_{12}, and B_3 with another state q_3. Of course the program associated with q_{12} has a conditional structure, depending on C_{12}, C_{21} and a Boolean variable. The program associated with q_3 is B_3.

Now, consider the mode-automaton M_2 that describes B_1 with a state q'_1, B_2 and B_3 with a state q'_{23}. In M_2, the program associated with q'_{23} has a conditional structure.

There is no relation R between the states of M_1 and the states of M_2 that would allow to recognize that they are equivalent. Translating them into mini-Lustre is a way of translating them into single-state mode-machines (with conditional associated programs), and it allows to show that they are indeed equivalent. On the other hand, if we are able to split q_{12} into two states, and q'_{23} into two states (as suggested in section 5.2) then the two machines have three states, and we can show that they are equivalent. □

Property 3 : Congruences of mode-automata
The two equivalences are congruences for parallel composition. □

7 Related Work and Comments on the Notion of *Mode*

We already explained in section 2.3 that there exists no construct dedicated to the expression of modes in the main synchronous programming languages. Mode-automata are a proposal for that. They allow to distinguish between *explicit* states (corresponding to modes, and described by the automaton part of the program) and *implicit* states (far more detailed and grouped into modes, described by the dataflow equational part of the program). This is an answer for people who argue that *modes* should not be related to *states*, because they are far more states than modes.

Other people argue that modes are not necessarily exclusive. The modes in one mode-automaton *are* exclusive. However, concurrent modes are elements of a Cartesian product, and can share something. Similarly, two submodes of a refined mode also share something. We tried to find a motivation for (and thus a definition of) *non-exclusive modes.*

Modecharts 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]. It is mentioned that *"the actual interaction with the environment is produced by the operations associated with entry and exit events"*. Hence the *modes* are not dealt with in the language itself; the language allows to describe a complex control structure, and an external activity can be attached to a composed state. It seems that the activity is not necessarily killed when the state is left; hence the activities associated with exclusive states are not necessarily exclusive. This seems to be the motivation for non-exclusive modes. Activities are similar to the external tasks of Esterel but, in Esterel, the way tasks interfere with the control struture is well defined in the language itself.

Real-time mode-machines have been proposed in [Pay96]. In this paper, modes are collections of states, in the sense recalled above (section 2.1). These collections are exhaustive but not exclusive. However, it seems that this requirement for non-exclusivity is related to *pipelining* of the execution: part of the system is still busy with a given piece of data, while another part is already using the next piece of data. The question is whether pipelining has anything to do with *overlapping* or *non-exclusive modes.* In software pipelining, there may be two components running in parallel and corresponding to the *same piece of source program*; if this portion of source describes modes, it may be the case that the two execution instances of it are in different modes at the same time, because one of them starts treating some piece of data, while the other one finishes treating another piece of data. Each instance is in exactly one mode at a given instant; should this phenomenon be called "non-exclusive modes"?

We are still searching for examples of *non-exclusive modes* in reactive systems.

8 Implementation and Further Work

We presented mode-automata, a way to deal with modes in a synchronous language. Parallel composition is well defined, and we also have a congruence of mode-automata, w.r.t. this operation. We still have to study the hierarchic composition, following the lines of [HMP95] (in this paper we proposed an extension of Argos dedicated to hybrid systems, as a description language for the tool Polka [HPR97], in which hybrid automata may be composed using the Argos operators. In particular, hierarchic composition in HybridArgos is a way to express that a set of states share the same description of the environment). We shall also extend the language of mode-automata by allowing full Lustre in the equations labeling the states (clocks, node calls, external functions or procedures...). Mode-automata composed in parallel are already available as a language, compiled into pure Lustre, thus benefiting from all the lustre tools.

We said in section 2 that *"It should be possible to project a program onto a given mode, and obtain the behaviour restricted to this mode"*. How can we do that for programs made of mode-automata? When a program is reduced to a single mode-automaton, the mode is a state, and extracting the subprogram of this mode consists in taking the equations associated with this state. The object we obtain is a mini-Lustre program without initial state. When the program is something more complex, we are still able to extract a non-initialized mini-Lustre program for a given composed mode; for instance the program of a parallel mode (q, q') is the union of the programs attached to q and q'.

Projecting the complete program onto its modes may be useful for generating efficient sequential code. Indeed, the mode-structure clearly identifies which parts of a program are active at a given instant. In the SCADE (Safety Critical

Application Development Environment) tool sold by Verilog S.A. (based upon a commercial Lustre compiler), designers use an *activation condition* if they want to express that some part of the dataflow program should not be computed at each instant. It is a low level mechanism, which has to be used carefully: the activation condition for subprogram P is a Boolean flow computed elsewhere in the dataflow program. Our mode structure is a higher level generalization of this simple mechanism. It is a real language feature, and it can be used for better code generation.

Another interesting point about the mode-structure of a program is the possibility of decomposing proofs. For the decomposition of a problem into concurrent tasks, and the usual parallel compositions that support this design, people have proposed *compositional* proof rules, for instance the assume-guarantee scheme. The idea is to prove properties separately for the components, and to infer some property of the global system. We claim that the decomposition into several modes — provided the language allows to deal with modes explicitly, i.e. to project a global program onto a given mode — should have a corresponding compositional proof rule. At least, a mode-automaton is a way of identifying a control structure in a complex program. It can be used for splitting the work in analysis tools like Polka [HPR97].

References

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

[BCH+85] J-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real time data-flow language. In *Real Time Systems Symposium*, San Diego, September 1985.

[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.

[CS95] C2A-SYNCHRON. The common format of synchronous languages – The declarative code DC version 1.0. Technical report, SYNCHRON project, October 1995.

[GBBG85] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gauthier. Signal: A data flow oriented language for signal processing. Technical report, IRISA report 246, IRISA, Rennes, France, 1985.

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

[HMP95] N. Halbwachs, F. Maraninchi, and Y. E. Proy. The railroad crossing problem, modeling with hybrid argos - analysis with polka. In *Second European Workshop on Real-Time and Hybrid Systems*, Grenoble (France), June 1995.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[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.

[JM88] Farnam Jahanian and Aloysius Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 14, 1988.

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

[MH96] F. Maraninchi and N. Halbwachs. Compiling argos into boolean equations. In *Formal Techniques for Real-Time and Fault Tolerance (FTRTFT)*, Uppsala (Sweden), September 1996. Springer verlag, LNCS 1135.

[MMP91] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In REX *Workshop on Real-Time: Theory in Practice*, DePlasmolen (Netherlands), June 1991. LNCS 600, Springer Verlag.

[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.