# MATOU: An Implementation of Mode–Automata*

Florence Maraninchi, Yann Rémond, and Yannick Raoul

VERIMAG – Joint Laboratory of Universit Joseph Fourier, CNRS and INPG
Centre Equation, 2 Av. de Vignate, F38610 GIERES
{Florence.Maraninchi,Yann.Remond}@imag.fr

**Abstract.** Mode-Automata have been proposed in [11]. They introduce, in the domain-specific data-flow language Lustre for reactive systems, a new construct devoted to the expression of *running modes*. The idea is to associate data-flow programs with the states of an automaton, representing modes. We define flat automata first, and then several composition operators, such as parallel composition and hierarchic composition, which give the language a state structure reminiscent from Statecharts. The semantics of this extension may be defined by describing the translation of Mode-automata into pure Lustre. However, the translation scheme is complex and it gives poor code; we study here the translation of mode-automata into the declarative format DC, used as an intermediate form in the compilers of several synchronous languages (Lustre, Esterel, ...). DC can be compiled into C, Java or Ada code. This allows to take advantage of the imperative mode-structure of a mode-automaton in order to improve the final sequential code.

## 1 Introduction

We are interested in *reactive* systems, which interact continuously with their environment. The *synchronous* approach [6] to the programming of reactive systems is represented by imperative languages like Esterel [2] and Argos [10], or by declarative data-flow languages like Signal [8] or Lustre [7]. 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. Recently, the users 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 data-flow language. Stategraphs were used, more or less, in order to represent the mode structure of the system.

In a data-flow language for reactive systems, both the inputs and outputs of the system are described by their *flows* of values along time. Time is discrete

* This work has been partially supported by Esprit LTR Project SYRF 22703

A. Watt (Ed.): CC/ETAPS 2000, LNCS 1781, pp. 249–263, 2000.
© Springer-Verlag Berlin Heidelberg 2000

and instants may be numbered by integers. If x is a flow, we will note $x_n$ its value at the $n$th reaction (or $n$th *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 "x = y + 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 describes a network of operators. 0ne should not write sets of equations with instantaneous loops, like : $\{x = y + z, z = x + 1, ...\}$. This is a set of fix point equations that perhaps has solutions, but it is not accepted as a data-flow program. For referencing the *past*, the operator `pre` is introduced : $\forall n > 0, (\texttt{pre}X)_n = X_{n-1}$. 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 previous value of T. Initialization of flows is provided by the `->` operator. The equation `X = 0 -> pre(X) + 1` defines the flow of integers.

In such a language, the notion of *running mode* corresponds to the fact that there may exist several equations for the same output, to be used in distinct periods of time. For instance, 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 data-flow constructs. Typically, some Boolean flows *mode1, mode2* are used to identify the current mode, and all other variables computed by the system may have definitions of the form: $X =$ *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, because it is hidden in the conditionals and the code for one mode is mixed up with the code dedicated to mode changes. This is exactly the same motivation as for the *state* design pattern [5] proposed for object-oriented designs; this pattern is used for allowing an object to alter its behavior when its internal state changes.

In object-oriented designs, the motivation for modes leads to a *pattern*, i.e. a recipe for writing a structured, modifiable and readable code, using the available constructs of the language. It is not compiled in a specific way. On the contrary, in the domain of safety-critical reactive systems, we would like the code we produce to benefit from the quite imperative structure implied by modes. Encoding modes with data-flow conditionals, even if it can be done in a structured and readable way, forbids efficient compilation. We need a new language *feature*, treated in a specific way by compilers, not only a *pattern*.

Starting from these ideas, we proposed the mathematical model of *mode-automata* [11], which can be viewed as a discrete form of *hybrid automata* [1]. In [11] we gave detailed motivations and related work on the notion of mode.

The present paper investigates implementation issues. First, we augment the formalism presented in [11] with a simple notion of hierarchic composition of mode-automata. Second, we define the semantics of mode-automata and their

compositions in terms of *activation conditions*, a notion available in the format DC [3] (for *"declarative code"*), used as an intermediate code in the compilers of several synchronous languages (Lustre, Esterel, Signal, Argos...).

Section 2 illustrates Lustre, flat mode-automata, DC and C for the same example. Section 3 recalls parallel composition, and defines hierarchic composition. In section 4, we show how to translate correct programs into DC, in a structural way. We give criteria for evaluating the quality of the generated code, and sketch the formal proof of the compilation scheme. Section 5 concludes, and draws some directions for further work.

## 2 An Example

### 2.1 Lustre, Mode-Automata and C

Figure 1 shows a simple Lustre program, a C program and a mode-automaton that have the same input/output behavior, illustrated by the timing diagrams. The reactive system inputs an integer `i` and outputs two integers `X` and `Y`. The Lustre program uses a Boolean memory `M` that commutes according to some conditions on `X`, and we can see that `X` and `Y` are updated depending on the value of `M`. This is a typical case where a mode-automaton can be useful.

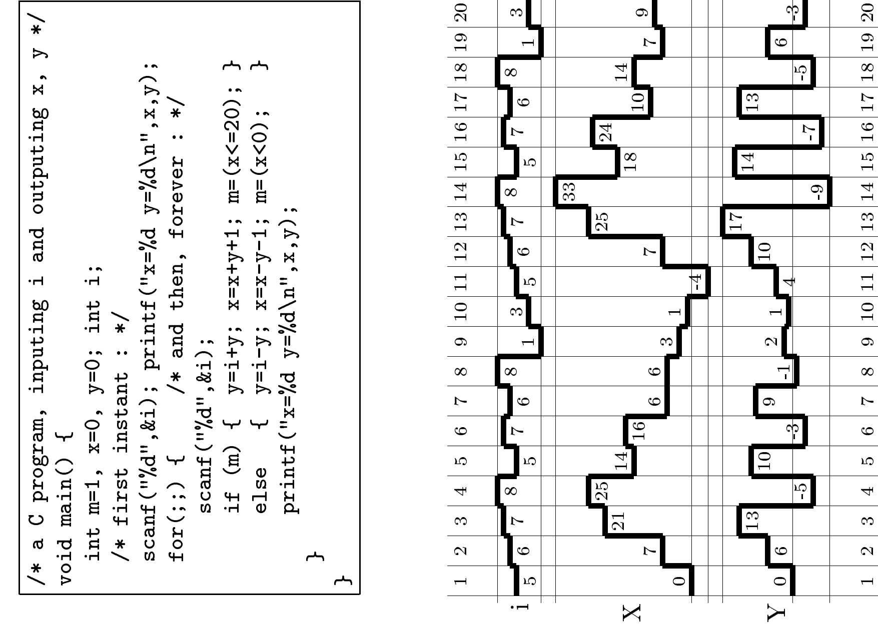
The mode-automaton we give here has two states, and equations attached to them. The transitions are labeled by conditions on `X`. The important point is that `X` and its memory are *global* to all states. The only thing that changes when the automaton changes states is the transition function; the memory is preserved. Hence, by construction, the behavior attached to the target state starts with the value of `X` that had been reached applying the equations attached to the source state. This gives the timing diagram of figure 1.

The C program is an infinite loop: this is the typical form of a sequential program produced from a synchronous language. However the code inside the loop has not been obtained automatically from the Lustre program. Indeed, in the example above, it could not: the IF conditional structure is *strict* in Lustre, as in a number of data-flow languages; the C program that corresponds to the Lustre program would compute both C expressions corresponding to `pre(X)+Y+1` and `pre(X)-Y-1` *before* choosing between the two for assigning a new value to `X`.

On the contrary, the C program we give here is relatively close to the one we would like to obtain from the mode-automaton. We would like the assignments to `x` and `y` to be *guarded* by an imperative conditional structure. Pieces of code attached to *inactive* modes should *not* be computed.

### 2.2 Clocks and States

In all data-flow synchronous languages, there exists a mechanism that allows to restrict the instants in which some flows are *defined*; this mechanism is usually called *clock* [4]. Associating clocks with the flows is an indirect way of controlling the instants in which the operators are indeed *computed*. For instance, in order to avoid a dynamic error like a division by zero, one has to use clocks.




```
-- A lustre program
node exemple_1 (i : int)  -- input
     returns  (X : int; Y : int); -- outputs
var M : bool; -- local variable
let
  M=true->if (pre(M) and (X>20)) or
            ((not pre(M)) and (X<0))
        then not pre(M) else pre(M);
  X=0->if pre(M) then pre(X) + Y + 1
          else pre(X) - Y - 1;
  Y=0->if pre(M) then i + pre(Y)
             else i - pre(Y);
tel
```

A Mode-automaton with input i and outputs X, Y

X init 0 : int
Y init 0 : int

A

B

X>20

X<0

X=pre(X)+Y+1
Y=i+pre(Y)

X=pre(X)-Y-1
Y=i-pre(Y)

**Fig. 1.** Example: Lustre, C, and Mode-Automata. The three programs have the same input/output behavior, described by the timing diagram (the horizontal axis is the discrete time; the values of the input $i$ are chosen arbitrarily.)

The execution of the Lustre program `X = if Y != 0 then U/Y else U` gives a dynamic error when `Y=0`, because the expression `U/Y` is computed before the choice that depends on `Y` being zero or not. Using clocks, one may write: `X = if Y != 0 then U/(Y when Y !=0) else U`; but this is a kind of typing error: all the operands of an operator should have the same clock. One then writes: `X = if Y != 0 then (U when Y !=0)/(Y when Y !=0) else U`. Then the same holds for `if`, which can be corrected by writing: `X = if Y != 0 then current ((U when Y !=0)/(Y when Y !=0)) else U ;` `current` is the oversampling operator; in this case, it gives values to the flow `X` even in the instants when `Y=0`. The semantics of sampling (`when`) and oversampling (`current`) ensures that the expression `U/Y` will be computed only when `Y` is not zero, which guarantees that there will be no dynamic error.

We were not happy with the translation of mode-automata into pure Lustre without clocks because we would like the states of a mode-automaton to behave as *clocks*, not as *strict conditional structures*. Hence we should translate mode-automata into Lustre *with clocks*, applying transformations like the one needed for the division, systematically. However, the semantics of clocks does not guarantee that the Lustre compiler be able to produce in all cases the ideal C program of the form `if (Y !=0) { X = U/Y ; } else { X = U ; }`.

### 2.3 The Intermediate Format DC

DC [3] has a declarative style, and provides an imperative mechanism called *activation condition*. Such conditions are Boolean flows that may be associated with basic operators or sub-networks, and allow to specify *when* things are computed. The `Lustre-to-DC` front-end translates clocks into activation conditions, and they are used in the back-end compilers (e.g. from DC to C), where they are translated into conditionals, guarding a set of assignments.

The two following 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`, whose evaluation is sequential.

| | |
|---|---|
| Equation defining X : | `X (init i) equcase: e1@a1, ..., ek@ak` |
| Memorization defining Y : | `Y (init i) memocase: e1@a1, ..., ek@ak` |

For equations:

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

For memorizations:

$$Y_0 = \mathtt{i}$$

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

We give below a DC program that has the same input/output behavior as the Lustre program and the mode-automaton of figure 1. Moreover, the equations attached to a state are computed only when necessary. The Boolean variable `M`

is used to encode the states of the mode-automaton, and serves as activation conditions. For instance, the flow `X` as a definition of the form:
`X equcase: 0@first; (MX+Y+1)@M; (MX-Y-1)@true`. Since the evaluation of activation conditions is sequential, `@true` means: `@(not M)`. The C program obtained from this DC program contains the following line, in which we recognize the structure of the ideal C program presented above: `if (first) {X=0;} else if(M) {X=MX+Y+1;} else {X=MX-Y-1;}`. This form is guaranteed by the semantics of DC activation conditions.

```
inputs:    i     int  ;  outputs:   X     int  ;   Y     int  ;
locals:    M     bool ;  first bool ; MX    int  ;   MY    int  ;
definitions:
  first (init true) memocase: false@true;
  MX    (init 0)    memocase:     X@true;
  MY    (init 0)    memocase:     Y@true;
  M     (init true) memocase: (X<=20)@M;       (X<0)@true;
  X                 equcase:      0@first;  (MX+Y+1)@M; (MX-Y-1)@true;
  Y                 equcase:      0@first;    (i+MY)@M;   (i-MY)@true;
```

### 2.4 Implementing Mode-Automata on Top of Lustre

For implementing Mode-Automata, either we translate them into Lustre with clocks, and then use the existing chain (Lustre to DC to C); or we translate them to DC (for translating them directly to C, we would have to rewrite part of the Lustre compiler, for the equations attached to states). Obtaining a Lustre program with clocks from a mode-automaton implies that the quite imperative structure of the mode-automaton be translated into the very declarative clock structure of Lustre... that has to be translated back to the imperative notion of activation condition. It is theoretically possible, but cumbersome to implement, especially when mode-automata are composed (see section 3.2 below). Moreover, keeping track of the interesting information about states along this path seems hard. We chose to translate mode-automata to DC. Producing DC code is simpler, and allows to use all the tools available for this format (formal verification, testing, debugging, etc.) without adding the Lustre intermediate form: source recovery is simpler. Moreover, DC is close to the internal formats of SCADE (the commercial version of Lustre, sold by Verilog S.A.), and the algorithms we give in this paper will be easy to reuse.

## 3 The Mode-Automata Language

### 3.1 Flat Mode-Automata: Formal Definition and Semantics

**Definition 1 (Mode-automata).** *Consider a set of* variables $\mathcal{V}$ *taking their values in a domain $D$, and a partial function $\mathcal{I} : \mathcal{V} \longrightarrow D$, used to define the* initial value *of some variables. We will note $\mathcal{V}_o = \mathsf{dom}(I)$ the set of* output *variables, and $\mathcal{V}_i = \mathcal{V} - \mathsf{dom}(I)$ the set of* input *variables. A mode-automaton on $\mathcal{V}$ is a tuple $(Q, q_0, f, T)$ where:*

- *$Q$ is the set of* states *of the automaton part and $q_0 \in Q$ is the* initial *state*
- *$T \subseteq Q \times C(\mathcal{V}) \times Q$ is the set of* transitions, *labeled by* conditions *on the variables of $\mathcal{V}$*
- *$f : \mathcal{V} \longrightarrow (Q \longrightarrow \mathsf{EqR}(\mathcal{V}))$ is a partial function ; a variable in $\mathcal{V}$ (typically an output) may be associated with a* total *function from $Q$ to the set $\mathsf{EqR}(\mathcal{V})$ of expressions that constitute right parts of the* equations *(not all variables are defined, but if a variable has an equation in one state, it has an equation in all states).*

$\mathsf{EqR}(\mathcal{V})$ *has the following syntax: $e ::= c \mid x \mid op(e, ..., e) \mid \texttt{pre}(x)$ where $c$ stands for constants, $x$ stands for a name in $\mathcal{V}$, and op stands for all combinational operators. The conditions in $C(\mathcal{V})$ are Boolean expressions of the same form, but without* `pre` *operators. The set of mode-automata is denoted by $\mathcal{M}$.*

Note that *Input* variables are intended to be used only in the right parts of the equations, or in the conditions. *Output* variables may be used everywhere. In the sequel, we use the domain $D = B \cup Z$ of Boolean and integer values, and we assume that all the expressions are typed correctly. We also assume that the equations attached to a state do not hide a cyclic dependency (like `X = Y ; Y = X ;` ); this is the usual Lustre criterion, which is used independently for each mode here. 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 $(q, c_2, q_2)$ and $q_1 \neq q_2$, 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 (however we usually omit some loops in the concrete syntax of mode-automata, as we did on the example of figure 1: the mode-automaton should show the loops $(A, X \leq 20, A)$ and $(B, X \geq 0, B)$).

Finally, the Lustre programs attached to states should not make use of the following operators: initialization (the initial value of variables is given globally), sampling and oversampling (states behave as implicit clocks). The conditions that label transitions do no make use of the `pre` operator.

**Definition 2 (Trace Semantics of Mode-automata).** *Consider a set of variables $\mathcal{V}$ and a partial initialization function $\mathcal{I}$. A input/output/state* trace *of a mode-automaton $M = (Q, q_0, f, T)$ on $\mathcal{V}$ is an infinite sequence $\alpha_n, n \in [0, +\infty]$ of tuples $\alpha_n = (i_n, o_n, s_n)$.*
*$\forall n, i_n$ (resp. $o_n$) is a valuation of the variables in $\mathcal{V} - \mathsf{dom}(I)$ (resp. $\mathsf{dom}(I)$), i.e. a total function $\mathcal{V} - \mathsf{dom}(I) \longrightarrow D$ (resp. $\mathsf{dom}(I) \longrightarrow D$) ; $s_n \in Q$. A trace $\sigma$ of such tuples is indeed a trace of $M$ if and only if:*

$$s_0 = q_0 \ \wedge\ \forall x \in \mathsf{dom}(I) \quad o_0(x) = \mathcal{I}(x)$$
$$(i) \wedge\ \forall (n > 0)\ \forall x \in \mathsf{dom}(I)$$
$$o_n(x) = f(x)(s_n)[i_n(z)/\mathtt{z}][i_{n-1}(z)/\mathtt{pre(z)}][o_n(y)/\mathtt{y}][o_{n-1}(y)/\mathtt{pre(y)}]$$
$$(ii) \wedge \forall (n \geq 0) \quad \exists (s_n, C, s_{n+1}) \in T \text{ such that: } C[i_n(z)/\mathtt{z}][o_n(y)/\mathtt{y}] = \mathtt{true}$$

In $(i)$ and $(ii)$ above, substitutions (denoted by [ ]) are done for all variables $z$ in $\mathcal{V} - \mathsf{dom}(I)$, and all variables $y$ in $\mathsf{dom}(I)$. Hence the occurrences of variable names are replaced by the current value of the variable, and the occurrences of

sub-expressions of the form `pre(x)` are replaced by the previous value of the variable. For all $n > 0$, this yields a circuit-free set of equations, of which the valuation of variables at instant $n$ is the unique solution.

### 3.2 Compositions

Figure 2 gives an example with parallel and hierarchic compositions. Their semantics can be given by showing how to obtain a trace-equivalent flat mode-automata from a composition of several flat automata. The compilation scheme does not follow this idea, however (see section 4).

Given two mode-automata $M1$ and $M2$ on a set $\mathcal{V}$, with the initialization function $\mathcal{I}$, and provided $\mathsf{dom}(f^1) \cap \mathsf{dom}(f^2) = \emptyset$, we denote their parallel composition by $M1 \times M2$. Its set of modes is the Cartesian product of the sets of modes of $M1$ and $M2$. The set of equations attached to a composed mode $A1A2$ (where $A1$ is a mode in $M1$ and $A2$ is a mode in $M2$) is the union of the equations attached to $A1$ in $M1$ and those attached to $A2$ in $M2$. The guard of a composed transition is the conjunction of the guards of the component transitions. The parallel composition of two mode-automata is correct if all the Lustre programs attached to the flat modes are correct: there is no instantaneous dependency loop, and each variable has exactly one equation.

The other composition is the *hierarchy* of modes. The sets of variables defined by the various mode-automata of the program are pairwise disjoint. In particular, a given variable $X$ may not be defined at several levels (see comments in the conclusion). This composition is described by the operation $\triangleright$, applied to a mode-automaton (not necessarily reactive) used as the overall controller, and a set of refining mode-automata: $\triangleright \;:\; \mathcal{M} \times 2^{\mathcal{M}} \longrightarrow \mathcal{M}$.

The equations attached to the refined state are distributed on all the substates; the transitions sourced in a refined state also apply to all the states inside; a transition that enters a refined state should go to the initial state (among all the states inside); a transition between two states inside may happen only if no transition from the refined state is firable (the outermost transitions have priority).

### 3.3 A Simple Language and Its Semantics

The set $\mathcal{E}$ of mode-automata expressions is defined by the following grammar, where NIL is introduced to express that a state is not refined and $M$ stands for a mode-automaton: $E ::= E \| E \quad | \quad \mathbf{R}_M(R_0, ..., R_n) \qquad R ::= E \mid \mathsf{NIL}$

The semantics of such a mode-automaton expression is a flat mode-automaton, obtained by applying the operations $\times$ and $\triangleright$ recursively ; since not all compositions are allowed, the semantic function may return the special error value $\perp$ ; if there is no composition error, the function returns a flat mode-automaton, which is both deterministic and reactive: $\mathcal{S} : \mathcal{E} \longrightarrow \mathcal{M} \cup \{\perp\}$. The recursive definition is given below (*null*, appearing below for NIL, is the function whose definition domain is empty).

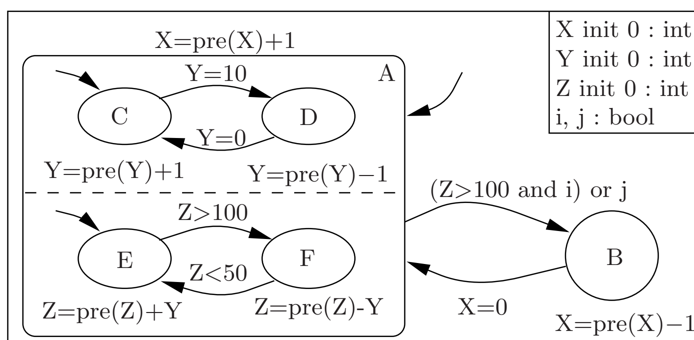



**Fig. 2.** A composition of mode-automata: parallel composition is denoted by a dashed line; hierarchic composition involves the main mode-automaton (with states A and B) and two refining sub-programs: the parallel composition in A, and nothing (NIL) in B. The states C, D, E and F are also "refined" by NIL. $Y$ and $Z$ are shared: they are computed by one of the mode-automata, and used as an input by another. The signals $i$ and $j$ are inputs. The corresponding expression is: $\mathbf{R}_{M_1}(\mathbf{R}_{M_2}(\mathsf{NIL}, \mathsf{NIL}) \| \mathbf{R}_{M_3}(\mathsf{NIL}, \mathsf{NIL}), \mathsf{NIL})$, where $M_1$ is the mode-automaton with states A and B, $M_2$ is the mode-automaton with states C and D, $M_3$ is the mode-automaton with states D and F.

```
 1 inputs:    i,j                                        : bool ;
 2 outputs:   X, Y, Z                                    : int ;
 3 locals:    first, kA, A, B, C', D', E', F', C, D, E, F : bool ;
 4            MX, MY, MZ                                 : int ;
 5 definitions:
 6  first (init true)  memocase: false      @true;
 7  X                  equcase:  0          @first; MX+1 @A;  MX-1 @B;
 8  Y                  equcase:  0          @first; MY+1 @C;  MY-1 @D;
 9  Z                  equcase:  0          @first; MZ+Y @E;  MZ-Y @F;
10  MX (init 0)        memocase: X          @true;
11  MY (init 0)        memocase: Y          @true;
12  M  (init 0)        memocase: Z          @true;
13  kA (init false) equcase:  (((Z>100 and i) or j) or false)  @A;
14  A  (init true)  memocase: not ((Z>100 and i) or j) @A; (X=0)@B;
15  B  (init false) memocase: not (X=0)@B; ((Z>100 and i) or j)@A;
16  C' (init true)  memocase:true @(kA or not A);not(Y=10) @C;  (Y=0)@D;
17  D' (init false) memocase:false@(kA or not A);not(Y=0)  @D; (Y=10)@C;
18  E' (init true)  memocase:true @(kA or not A);not(Z>100)@E; (Z<50)@F;
19  F' (init false) memocase:false@(kA or not A);not(Z<50) @F;(Z>100)@E;
20  C                  equcase: false@not A;    C'      @true;
21  D                  equcase: false@not A;    D'      @true;
22  E                  equcase: false@not A;    E'      @true;
23  F                  equcase: false@not A;    F'      @true;
```

**Fig. 3.** The DC program obtained for the program of figure 2

$$\mathcal{S}(E_1 \| E_2) = \begin{cases} \bot \text{ if } \mathcal{S}(E_1) = \bot \text{ or } \mathcal{S}(E_2) = \bot \text{ or } \mathsf{dom}(\mathcal{S}(E_1).f) \cap \mathsf{dom}(\mathcal{S}(E_1).f) \neq \emptyset \\ \mathcal{S}(E_1) \times \mathcal{S}(E_2) \text{ otherwise} \end{cases}$$

$$\mathcal{S}(\mathbf{R}_M(R_0, ..., R_n)) = \begin{cases} \bot \text{ if } \exists i \in [0, n] \text{ s.t. } \begin{cases} \mathcal{S}(R_i) = \bot \\ \text{ or } \mathsf{dom}(\mathcal{S}(R_i).f) \cap \mathsf{dom}(M.f) \neq \emptyset \end{cases} \\ \quad \text{or } \exists i, j \in [0, n]. i \neq j \wedge \mathsf{dom}(\mathcal{S}(R_i).f) \cap \mathsf{dom}(\mathcal{S}(R_j).f) \neq \emptyset \\ M \triangleright (\mathcal{S}(R_1), ...\mathcal{S}(R_n)) \text{ otherwise} \end{cases}$$

$$\mathcal{S}(\mathsf{NIL}) = (\{\mathsf{NIL}\}, \mathsf{NIL}, \textit{null}, \{(\mathsf{NIL}, \mathsf{true}, \mathsf{NIL})\})$$

A *Mode-automaton program* is a tuple $(\mathcal{V}, \mathcal{I}, E)$ where $\mathcal{V}$ is a set of variables, $\mathcal{I} : V \longrightarrow D$ an initialization (partial) function, and $E$ a mode-automaton expression as defined above, composed from mode-automata on $\mathcal{V}$. $\mathcal{V}$ and $\mathcal{I}$ play the role of a set of *global variable declarations*, for all variables appearing in the mode-automata of the expression $E$. The set of mode-automata programs is denoted by $\mathcal{P}$. A mode-automata program $(\mathcal{V}, \mathcal{I}, E)$ is said to be *correct* if and only if $\mathcal{S}(E) \neq \bot$ and $\mathsf{dom}(\mathcal{S}(E).f) = \mathsf{dom}(I)$, i.e. there are no internal conflicts in $E$ and the declarations are consistent with the use of the variables in $E$.

## 4 Implementation by Translation into DC

A DC program is a tuple $(I, O, L, Mm, Eq)$ where $I$, $O$ and $L$ are the sets of input, output and local variables, $Mm$ is the set of memorizations and $Eq$ is the set of equations. Each memorization or equation is a tuple $< v, i, \sigma >$ where $v$ is the name of the variable being defined, $i$ is the initial value and $\sigma$ is a *sequence* (denoted by $[]$) of (expression@activation)s. The set of DC programs is denoted by $\mathcal{DC}$. Our translation is defined in order to guarantee the following:

**Property 1** *(Form of the* DC *code)*
The DC code corresponding to the equations attached to a global state $X$, and to the conditions of the transitions sourced in $X$, are computed exactly when this state $X$ is active.

For the typical programs we have in mind (a few modes, and big programs attached to modes), this is our notion of *good* code.

### 4.1 The Translation Algorithm

**The Translation Function** $\Gamma : \mathcal{P} \longrightarrow \mathcal{DC}$ is the function translating main mode-automata programs into DC programs. It is defined on top of the $\gamma$ function (to be defined later). For a correct (see paragraph 3.3) program $(\mathcal{V}, \mathcal{I}, E)$:

$$\begin{aligned} \Gamma((\mathcal{V}, \mathcal{I}, E)) = \ & \textbf{let} \ \ (L, Eq, Mm) = \gamma(\mathcal{I}, E, \texttt{false}, \texttt{true}) \\ & \textbf{in} \ \ (\mathcal{V} - \mathsf{dom}(I), \ \mathsf{dom}(I), \ L \cup \{\texttt{first}\}, \ Eq, \\ & \qquad Mm \cup \{ \ < \texttt{first}, \ \texttt{true}, \ [\texttt{false@true}] > \ \}) \end{aligned}$$

Figure 3 gives the DC program obtained for the example of figure 2. It always contains the Boolean variable `first`, which is true at the first instant,

and then false forever. It is defined by the the memorization $<$ `first`, `true`, [`false@true`] $>$ (line 6). It is used as an activation condition for the encoding of other variables.

$\gamma$ is the interesting function. It takes an initialization function $\mathcal{I}$, a mode-automaton expression $E$ and two DC Boolean expressions $k$ (for "*kill*") and $a$ (for "*alive*"), and it gives a tuple $(L, Eq, Mm)$, where $L$ is a set of fresh variables needed for encoding the expression $E$ into DC, $Eq$ and $Mm$ are the equations and memorizations obtained by encoding $E$. The variables in $L$ are intended to become *local* variables of the DC program.

The two expressions $k$ and $a$ are used to encode a sub-expression depending on its context ; they are inherited parameters in the recursive definition of $\gamma$, starting from the root of the expression tree with values $k =$ `false` (the main program is never killed) and $a =$ `true` (the main program is always alive).

Since the parameter $\mathcal{I}$ of $\gamma$ is transmitted unchanged through the recursive calls, we omit it as an explicit parameter, in the following definitions.

**NIL and Parallel Composition** There is little to do in order to encode the NIL leaves. $k$ and $a$ are transmitted unchanged to the operands of a parallel composition, and the global DC program is obtained by gathering the sets of fresh variables, equations and memorizations introduced for encoding the two operands:

$$\begin{aligned}
\gamma(\mathsf{NIL}, k, a) &= (\emptyset, \emptyset, \emptyset) \\
\gamma(E_1 \| E_2, k, a) &= \textbf{let} \;\; (L_1, Eq_1, Mm_1) = \gamma(E_1, k, a) \\
&\qquad\quad\; (L_2, Eq_2, Mm_2) = \gamma(E_2, k, a) \\
&\quad\;\; \textbf{in} \;\; (L_1 \cup L_2,\; Eq_1 \cup Eq_2,\; Mm_1 \cup Mm_2)
\end{aligned}$$

**Hierarchic Composition** The interesting (and difficult) case is the hierarchic composition: we encode the states and transitions of the main automaton, and the *kill* and *alive* parameters transmitted to the refining mode-automata are built according to the Boolean DC variables encoding the states of this main mode-automaton.

The idea is the following: we introduce 3 fresh Boolean variables $(s_i, s_i', k_i)$ per state of the refined mode-automaton. $\epsilon$ is a set of equations defining the $s_i$ variables. $s_i$ means: "*the global program is currently in the state $s_i$ and in all of its ancestors*" ; it is defined as being `false` when the context says that the automaton is not alive, otherwise it copies the value of the other state variable $s_i'$. $s_i'$ is defined by a memorization in $\mu$, which corresponds to the classical encoding of an automaton, adapted to the DC style with activation conditions. Moreover $s_i'$ is set to its initial value whenever $k \vee \neg a$, i.e. when the automaton is being killed, or is currently not alive (see Figure 3, lines 14-23. the example is optimized a little: for the states belonging to the upper level, we do not need two state variables, and we use only one).

$$\begin{aligned}
\epsilon &= \{ < s_i,\; ?,\; [\, \text{false@}(\neg a),\; s_i'\text{@true} \,] > \}_{i \in [0,n]} \\
\mu &= \{ < s_i',\; \square,\; [\, \square\text{@}(k \vee \neg a),\; (\neg \textstyle\bigvee_{(q_i, C, q_j) \in T} C)\text{@}s_i \,] > \}_{i \in [0,n]}
\end{aligned}$$

("?" means any value may be chosen, because it is never used; $\square$ is "`true`" if $i = 0$ (initial state) and "`false`" otherwise).

$k_i$ means "*the mode-automaton refining state number $i$ has to be killed*" and is defined by an equation in $\chi$ ; its definition shows two cases: either the global program is killed ($k$), or the state $i$ is left because one of the outgoing transition conditions is true (see Figure 3, line 13).

$$\chi = \{< k_i,\ \text{false},\ [\ (k \vee \bigvee_{(q_i,C,q_j)\in T} C)\texttt{@}s_i\ ] >\}_{i\in[0,n]}$$

Encoding the refining program $R_i$ with $s_i$ as the alive parameter and $k_i$ as the kill parameter, gives the $(rL_i, rEq_i, rMm_i)$, and encoding the equations attached to the states of the refined mode-automaton, gives the $(XEq_i, XMm_i, XL_i)$. All the sets of fresh variables, equations and memorizations are then gathered.

$$\begin{array}{l}
\gamma(\ \mathbf{R}_{(Q,q_0,f,T)}(R_0, ..., R_n)\ , k, a) = \\
\textbf{let}\ \ \Omega = \{s_0, ..., s_n,\ s'_0, ..., s'_n,\ k_0, ..., k_n\}, \quad \epsilon = ..., \chi = ..., \mu = ...\ \text{(see above)} \\
\quad (rL_i, rEq_i, rMm_i) = \boxed{\gamma(R_i, s_i, k_i)}, \quad i \in [0, n] \qquad \textit{(refining programs)} \\
\quad (XEq_i, XMm_i, XL_i) = \boxed{\delta(Q, f, \mathsf{dom}(f))} \qquad \textit{(refined mode-automaton)} \\
\textbf{in}\ (\bigcup_{i\in[0,n]} rL_i \cup \bigcup_{i\in[0,n]} XL_i \cup \Omega, \quad \bigcup_{i\in[0,n]} rEq_i \cup \bigcup_{i\in[0,n]} XEq_i \cup \epsilon \cup \chi, \\
\quad \bigcup_{i\in[0,n]} rMm_i \cup \bigcup_{i\in[0,n]} XMm_i \cup \mu)
\end{array}$$

**Encoding Lustre Equations** $\delta$ takes the set $Q$ of states of the main automaton, the function $f$ that associates definitions of variables with the states, and the set $\mathsf{dom}(f)$ of variables for which we need to generate DC definitions.

For each variable $v \in \mathsf{dom}(f)$, $\delta$ gathers the DC expressions $e_i, i \in [0, n]$ obtained by translating (with the function $\theta$ given below) the Lustre equations attached to states $(f(v)(q_i), i \in [0, n])$ into a *single equation* $< v, ?, [\mathcal{I}(V)\texttt{@first}, e_0\texttt{@}s_0, ..., e_n\texttt{@}s_n] >$, adding a case for the initial instant (see Figure 3, lines 7-9). The order of the cases is irrelevant, since the $s_i$ Boolean variables are guaranteed to be pairwise exclusive (they encode the states of the main automaton).

$$\begin{array}{l}
\delta(Q, f, \{v\} \cup V) = \textbf{let}\ \ (e_i, Mm_i, L_i) = \theta(f(v)(q_i)), \quad i \in [0, n] \\
\qquad\qquad\qquad\qquad (Eq', Mm', L') = \delta(Q, f, V) \\
\qquad\qquad\qquad \textbf{in}\ (Eq' \cup \{< v, ?, [\ \mathcal{I}(V)\texttt{@first}), e_0\texttt{@}s_0, ...e_n\texttt{@}s_n\ ] >\}, \\
\qquad\qquad\qquad\quad Mm' \cup \bigcup_{i\in[0,n]} Mm_i, \quad L' \cup \bigcup_{i\in[0,n]} L_i) \\
\quad \delta(Q, f, \emptyset) = (\emptyset, \emptyset, \emptyset)
\end{array}$$

**Translation of Lustre Into DC** The $\theta$ function translates the Lustre expressions attached to states of the mode-automata into DC expressions, possibly creating variables and memorizations (for the `pre` sub-expressions); it returns a tuple $(e, Mm, L)$ where $e$ is a DC expression, $Mm$ is a set of memorizations and $L$ is a set of new variables, to be added to the set of local variables of the global DC program. We define $\theta$ for binary operators (expr$_1$ op expr$_2$), variables (v),

constants (cst) and `pre` sub-expressions. Lustre constants and operators have a counterpart in DC.

$$\begin{aligned}
\theta(\text{expr}_1 \text{ op } \text{expr}_2) &= \textbf{let } \ (e_1, Mm_1, L_1) = \theta(\text{expr}_1), \quad (e_2, Mm_2, L_2) = \theta(\text{expr}_2) \\
&\quad\ \textbf{in} \quad (e_1 \text{ op } e_2,\ Mm_1 \cup Mm_2,\ L_1 \cup L_2) \\
\theta(v) &= (v, \emptyset, \emptyset) \qquad\qquad \theta(\text{cst}) \ = \ (\text{cst}, \emptyset, \emptyset) \\
\theta(\texttt{pre}(v)) &= \textbf{let } Mv \text{ be a fresh variable} \\
&\quad\ \textbf{in} \quad (Mv,\ \{< Mv, \mathcal{I}(v), [v@\text{true}] >\},\ \{Mv\})
\end{aligned}$$

The last line means that `Mv` is the memory of `v`, initialized as required in the global function $\mathcal{I}$ of the program, and always computed (@true). If there are several occurrences of `pre(v)`, the variable `Mv` and its definition are created only once (see Figure 3, lines 10-12).

### 4.2 Correctness of the Translation Scheme

Both mode-automata and DC have a formal trace semantics, i.e. there exists a mathematically defined function $f_m$ from mode-automata to input/output traces, and another function $f_d$ from DC programs to input/output traces. Traces are sets of sequences of input/output tuples. We have to prove that :

$$\forall P = (\mathcal{V}, \mathcal{I}, E).\ \ f_m(\mathcal{S}(P)) = f_d(\gamma(\mathcal{I}, E, \texttt{false}, \texttt{true}))$$

where $P = (\mathcal{V}, \mathcal{I}, E)$ is a mode-automaton program as defined in section 3.3.

However, since the translation algorithm does not perform complex optimizations like minimizing the automaton structure, $\mathcal{S}(P)$ and $\gamma(\mathcal{I}, E, \texttt{false}, \texttt{true})$ are more than trace-equivalent: they are *isomorphic*, which is easier to prove.

We extend the semantics of mode-automata to input/output/state traces, and that of DC to input/output/local traces. We then exhibit a one-to-one function $\lambda$ relating a global state of a mode-automaton program with a configuration of the local Boolean variables used for encoding states in the DC program. Then we have to prove that: first, $\mathcal{S}(P)$ and $\gamma(\mathcal{I}, E, \texttt{false}, \texttt{true})$ have the same initial state (via $\lambda$); second, if we start from a global state (of the mode-automaton program) and a configuration of the variables (of the DC program) related by $\lambda$, and take the same input into account, then the two objects produce the same output and evolve to a new global state and a new configuration of DC variables that are, again, related by $\lambda$. This is sufficient for proving that $\mathcal{S}(P)$ and $\gamma(\mathcal{I}, E, \texttt{false}, \texttt{true})$ have the same sets of input/output traces.

### 4.3 Quality of the Translation Scheme

We already said that the typical systems we have in mind have a few modes, and big programs attached to modes (this is not the case in our example, for sake of simplicity, but imagine that we replace `X = pre(X) − 1` by a one-page program). Our criterion for *good code* is the property 1, page 258. Our translation scheme guarantees it, due to the careful encoding of states with two variables $s$ and $s'$ (except at the upper level which is never killed).

We could also take the number of variables into account, for this is the main parameter that plays a role in the complexity of the static analysis algorithms that could be applied at the DC level. Reducing the number of variables was not our main aim but, yet, the encoding is not so bad: a log encoding of the states of a single mode-automaton into Boolean DC variables would make the transitions very complex for a very little gain, because the automaton components are supposed to be small. The structural encoding of composed programs ensures that *global* states are encoded in an efficient way. Since there exist optimizations techniques at the DC level, we should concentrate on the optimizations that can be performed only at the mode-automaton level. For instance, we could use the hierarchic structure of a mode-automaton program in order to *reuse* some DC variables used for the encoding of states.

## 5 Conclusions and Future Work

The algorithm presented in this paper has been implemented in the tool MATOU by Yann Rémond, on top of the DRAC set of tools for the DC format, developed at Verimag by Y. Raoul, and part of the SYRF project [12]. MATOU has been used for several case studies, among which: a simplified temperature control system for an aircraft, submitted by SAAB and already studied in the SYRF [12] project; the production cell [9] that was proposed at FZI (Karlsruhe) as a test bench for several languages and proofs tools; an operational nuclear plant controller submitted by Schneider Electric. These three examples fall in the category we are interested in: a little number of modes, and quite complex programs for each mode. The code we obtained is satisfactory, but we still need to run MATOU on a test-bench, for determining where optimizations should be applied.

Concerning the language constructs, the equations attached to states are written using a very small subset of Lustre, sufficient for demonstrating the interest of mode-automata, and for which we can perform the compilation into DC in a simple way. Our notion of hierarchic composition is also simple; in particular, we reject programs in which the same variable is defined at several levels. We are working on more permissive definitions of the hierarchic composition, inspired by some medium-size examples, in which such a situation is allowed, and treated like some special case of inheritance. However, as far as the translation into DC is concerned, the algorithm described in this paper will continue to be the basis of the implementation. For the moment, it seems that the more advanced versions of the constructs will be implemented by some transformations of the abstract tree, before generating DC code.

Further work on the definition and implementation of mode-automata includes some easy extensions (variables local to a state and its outgoing transitions; priorities between transitions sourced in the same state, importing objects from a host language like C, ...) and some extensions of the subset of Lustre we allow to label states (calling nodes with memory, using Lustre clocks, ...) that

require a careful study of the interaction between the automaton structure and the Lustre constructs.

Finally, concerning the translation scheme, we managed to take the mode-structure into account fully: we avoid unnecessary computations. DC turns out to be the appropriate level for studying the introduction of an imperative construct into a data-flow language: the format is still declarative and equational, but the notion of activation condition gives a pretty good control on the final C code. Moreover, although the implementation is particular to the precise structure of DC code, we think that the ideas developed in the paper can be reused for translating modes into a wide variety of formats; for instance, we plan to study the translation of Mode-Automata into other target codes like VHDL or VERILOG. The method can also be used to introduce modes in other data-flow languages. Moreover, DC is close to SCADE and the translation of mode-automata into one of the SCADE internal formats does not bring new semantical problems.

## References


1. 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. 250
2. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992. 249
3. C2A-SYNCHRON. The common format of synchronous languages – The declarative code DC version 1.0. Technical report, SYNCHRON project, October 1995. 251, 253
4. P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992. 251
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995. 250
6. N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993. 249
7. 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. 249
8. 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. 249
9. Claus Lewerentz and Thomas Lindner. *Formal Development of Reactive Systems: Case Study Production Cell*. Number 891 in Lecture Notes in Computer Science. Springer Verlag, January 1995. 262
10. F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*. LNCS 630, Springer Verlag, August 1992. 249
11. 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. 249, 250
12. SYRF. Esprit ltr 22703, "synchronous reactive formalisms". Technical report, 1996-1999. `http://www-verimag.imag.fr/SYNCHRONE/SYRF/syrf.html`. 262