Generating Efficient Code From Data-Flow Programs *

Nicolas Halbwachs , Pascal Raymond
IMAG - LGI (U.A. CNRS 398)
B.P.53X, 38041 Grenoble, France

Christophe Ratel
Merlin Gerin - SES
38050 Grenoble, France

Abstract

This paper presents the techniques applied in compiling the synchronous data-flow language LUSTRE. The most original technique consists in synthesising an efficient control structure, by simulating the behavior of boolean variables at compile-time. Here, the techniques are explained on a small subset of LUSTRE.

1 Introduction

Many authors [Kah74, Gra82, PP83, AW85] have advocated the advantages of data-flow languages, mainly due to their mathematical soundness, the ease of formal program construction and transformation and the absence of side effects. However, no such language is actually used, mainly because no good compilers exist for standard machines. The absence of assignment and control structures makes it difficult to produce efficient code from a data-flow program.

We have argued elsewhere [BCHP86, CPHP87] that the declarative style allowed by data-flow languages makes them especially suitable for a class of real time programs: such programs can be found in domains (automatic control, signal processing, hardware simulation,...) where traditional tools are declarative (differential equations, operator networks,...). However, in these domains, the need for efficient code is even more important than in usual programming. On the other hand, in order to adapt data-flow languages to real time programming, we have proposed that they be given a synchronous interpretation: each operator receives all of its inputs at the same time, and all the operators synchronously respond to their inputs. Such an interpretation restricts the behavior of a program in such a way that the production of efficient code becomes possible, through a static synthesis of control structures, as done by the compiler of the imperative synchronous language ESTEREL [BG92]. The synchronous data-flow language LUSTRE [CPHP87] and its compiler are based on these ideas.

*This work was partially supported by ESPRIT-BRA Project "SPEC", by PRC-C^3 (CNRS) and by a contract from Merlin-Gerin

This paper deals with this compilation process, illustrated on a very simple language, which is a subset of Lustre and is introduced in section 2. In section 3, we show the simplest way for compiling a program in this language, as a single loop. Section 4 explains how a program may be partially simulated, and turned into a finite automaton representing the control skeleton of the object code: this automaton is obtained by a static simulation of the boolean variables involved in the program: in our language, boolean variables are used to model what is usually expressed through control structures in imperative languages. However, experience shows that this exhaustive simulation often produces an automaton far from minimal; this phenomenon, which is less apparent in the case of an imperative language like Esterel, comes from the fact that the declarative style encourages the definition of variables regardless of their effective use: one writes equations which are assumed to be always true; there are no control structures to indicate that some variables are only intermittently used, and the fact that they are always computed creates many irrelevant states. We shall therefore concentrate on a kind of "demand driven simulation", which consists of computing, in a given state, only those boolean variables which are involved in the computation either of the output or of the next state. In [BFH90], we have proposed a general algorithm for generating a minimal automaton from a finite state program. Section 5 presents the adaptation of this algorithm for compiling our language. In conclusion, we shall give some experimental results about the implementation of the method, which has been done in the compiler Lustre-V3.

2 A very simple data-flow language

2.1 Informal presentation

For the sake of simplicity, we shall illustrate the compilation process on a small subset of Lustre. As usual in the data-flow approach, this language operates over infinite sequences of values: any variable or constant is equated to the sequence of values it takes during the execution of the program, as the operators are intended to operate globally over sequences. The synchronous interpretation consists of considering that a program has a cyclic behavior, consisting of computing at the n-th cycle the n-th value of each variable (this strict synchrony may be released in full Lustre). A program is a set of variable definitions, expressed by a system of equations. The equation "`X=E`", where `X` is a variable representing the sequence $(x_1, x_2, \ldots, x_n, \ldots)$ and `E` is an expression representing the sequence $(e_1, e_2, \ldots, e_n, \ldots)$, means that $x_n = e_n$, for any positive integer n.

Expressions are built using constants (infinite constant sequences), variables and operators. Usual arithmetic, boolean and conditional operators are extended to pointwisely operate over sequences, and are hereafter referred to as "data operators". For instance, the expression "`if X>=Y then X-Y else Y-X`" represents the sequence whose n-th term is the absolute difference of the n-th terms of the sequences represented by `X` and `Y`.

In addition to those data operators, the language only contains one specific operator: the "`pre`" operator (for "previous"). If `E` is an expression representing the sequence $(e_1, e_2, \ldots, e_n, \ldots)$, and `initial` is a constant of the same type as `E`, then `pre(E,initial)` represents the sequence $(\texttt{initial}, e_1, e_2, \ldots, e_{n-1}, \ldots)$.

We are now able to define non-trivial sequences, by means of recursive definitions. For instance, the equation "`N = pre(N + 1, 0)`" defines `N` to be the sequence of naturals, and "`C = not pre(C, false)`" defines `C` to be the alternating sequence $(\mathit{true}, \mathit{false}, \mathit{true}, \mathit{false}, \ldots)$.

2.2 Formal semantics

Let us define the operational semantics of our language, which will be the basis of the compilation process. We define a memory σ to be a function from identifiers to values, and an history η to be an infinite sequence of memories. Given an history of input variables of a program, the semantics will provide the whole history of all variables. The semantics are described by means of structural inference rules [Plo81], which define the following predicates:

- $\eta \vdash \mathit{eqs} : \eta'$ which means "given the input history η, the program consisting of the system of equations *eqs* defines the global history η' "
- $\mathit{eq} \xrightarrow{\sigma} \mathit{eq}'$ which means "the right-hand side of the equation *eq* evaluates as $\sigma(\mathtt{X})$, where `X` is its left-hand side identifier, and *eq* will be later on considered as *eq'* ". If this predicate holds, *eq* will be said *compatible* with σ.
- $\sigma \vdash e \xrightarrow{v} e'$ which means "the expression e evaluates as v in the memory σ, and will be later on considered as e'"

Here are the rules, which are commented below:

$$\frac{\mathit{eqs} \xrightarrow{\sigma} \mathit{eqs}' \quad , \quad \eta \vdash \mathit{eqs}' : \eta'}{\sigma_{in}.\eta \vdash \mathit{eqs} : \sigma.\eta'} \;(1) \qquad \frac{\mathit{eq} \xrightarrow{\sigma} \mathit{eq}' \quad , \quad \mathit{eqs} \xrightarrow{\sigma} \mathit{eqs}'}{(\mathit{eq}; \mathit{eqs}) \xrightarrow{\sigma} (\mathit{eq}'; \mathit{eqs}')} \;(2) \qquad \frac{\sigma \vdash \mathtt{E} \xrightarrow{\sigma(\mathtt{X})} \mathtt{E}'}{\mathtt{X=E} \xrightarrow{\sigma} \mathtt{X=E}'} \;(3)$$

$$\frac{\sigma \vdash \mathtt{E}_0 \xrightarrow{v_0} \mathtt{E}_0' \quad , \quad \sigma \vdash \mathtt{E}_1 \xrightarrow{v_1} \mathtt{E}_1'}{\sigma \vdash \mathtt{E}_0 \;\mathtt{or}\; \mathtt{E}_1 \xrightarrow{v_0 \vee v_1} \mathtt{E}_0' \;\mathtt{or}\; \mathtt{E}_1'} \;(4) \qquad \frac{\sigma \vdash \mathtt{E} \xrightarrow{w} \mathtt{E}'}{\sigma \vdash \mathtt{pre(E,}v\mathtt{)} \xrightarrow{v} \mathtt{pre(E',}w\mathtt{)}} \;(5)$$

(1) Rule of programs: If the program *eqs* is compatible with the memory σ and must be later on be considered as *eqs'*, and if *eqs'* transforms the history η into η', then *eqs* transforms the history $\sigma_{in}.\eta$ into $\sigma.\eta'$, where σ_{in} denotes the restriction of σ to the inputs variables of the program, and "." denotes the concatenation on histories.

(2) Systems of equations: A system of equations is compatible with a memory σ if and only if each equation is compatible with σ. Each equation is rewritten for further evaluation.

(3) Equations: An equation `X = E` is compatible with σ if and only if its right-hand side evaluates as $\sigma(\mathtt{X})$ in σ. Its right hand-side is rewritten for further evaluation.

(4,5) Expressions: Any n-ary data operator `op` evaluates always in the same way: The rule 4 concerns the boolean operator `or` (rules for other data operators are similar). There is a special rule (5) for the operator `pre`: it always evaluates as it second operand, and the current value of its first operand is "stored" by the rewriting, to be returned at the next evaluation.

For instance, we can apply these rules to get the two first steps of the behavior of the "program" `C = not pre(C, false)` (we note $[v/\mathtt{C}]$ the memory associating v to `C`):

$$\dfrac{\dfrac{\dfrac{[true/\mathtt{C}] \vdash \mathtt{C} \xrightarrow{true} \mathtt{C}}{[true/\mathtt{C}] \vdash \mathtt{pre(C,\ false)} \xrightarrow{false} \mathtt{pre(C,\ true)}}}{[true/\mathtt{C}] \vdash \mathtt{not\ pre(C,\ false)} \xrightarrow{true} \mathtt{not\ pre(C,\ true)}}}{\mathtt{C\ =\ not\ pre(C,\ false)} \xrightarrow{[true/\mathtt{C}]} \mathtt{C\ =\ not\ pre(C,\ true)}}$$

$$\dfrac{\dfrac{\dfrac{[false/\mathtt{C}] \vdash \mathtt{C} \xrightarrow{false} \mathtt{C}}{[false/\mathtt{C}] \vdash \mathtt{pre(C,\ true)} \xrightarrow{true} \mathtt{pre(C,\ false)}}}{[false/\mathtt{C}] \vdash \mathtt{not\ pre(C,\ true)} \xrightarrow{false} \mathtt{not\ pre(C,\ false)}}}{\mathtt{C\ =\ not\ pre(C,\ true)} \xrightarrow{[false/\mathtt{C}]} \mathtt{C\ =\ not\ pre(C,\ false)}}$$

3 Computation order and single loop

The simplest code we can generate from a program written in our language consists of a global infinite loop computing all the variables of the program, in suitable order. The existence of such an order must be considered first: The current value of a variable may only depend on its past values, since we don't intend to give sense to implicit definitions like `X=X*X-1`. The existence of a computation order comes down to the irreflexivity of some dependency relation among variables. Several dependency relations can be considered, which lead to different acceptance criteria; here are two examples:

- *Semantic dependency:* An expression `E` in a program *eqs semantically depends* on a variable `X` if and only if there exists a sequence of memories $(\sigma_0, \ldots, \sigma_n)$, a memory σ and a value w such that $eqs \xrightarrow{\sigma_0} eqs_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} eqs_n$, $\sigma \vdash \mathtt{E}_n \xrightarrow{v} \mathtt{E}'$ and $\sigma[w/\mathtt{X}] \not\vdash \mathtt{E}_n \xrightarrow{v} \mathtt{E}'$, where $\mathtt{E}_n$ is the expression such that $\sigma_0 \vdash \mathtt{E} \xrightarrow{v_0} \mathtt{E}_0 \ldots \sigma_n \vdash \mathtt{E}_{n-1} \xrightarrow{v_n} \mathtt{E}_n$ and $\sigma[w/\mathtt{X}]$ denotes the memory $\lambda\mathtt{Y}.$ if $\mathtt{Y} \equiv \mathtt{X}$ then w else $\sigma(\mathtt{Y})$.

  The semantic dependency is the most precise we can define: A variable `X` semantically depends on `Y` if and only if there is a behavior of the program leading to a situation where the value of `X` can depend of the current value of `Y`. The problem is that deciding whether a variable semantically depends on itself is generally undecidable. So, we need a stronger acceptance criterion.

- *Syntactic dependency:* In a given program *eqs*, a variable `X`, defined by the equation `X = E` *directly depends* on a variable `Y` if and only if `Y` appears outside any `pre` operator in `E`. Let us define the *syntactic dependency* relation to be the transitive closure of that direct dependency. Deciding of the irreflexivity of this relation is straightforward. Of course, an acceptance criterion based on the syntactic dependency will reject some meaningful programs, like the following:

```
X = if C then Y else Z ; Y = if C then Z else X
```

since `X` syntacticly depends on `C,Y,Z` and `X` (since `Y` depends on `X`).

If the program does not contain dependency loops, any total order compatible with the dependency order is a suitable computation order. The choice of a particular order can influence the size of necessary memory. Generally speaking, the result of a `pre` operator must be stored in an auxiliary variable, in order to be available at the next cycle. However, this auxiliary memory can often be saved: Consider an expression `pre(X,`v`)`, and assume that all the variables depending on that expression can be ordered before `X`. Then, there is no need of auxiliary memory for storing the previous value of `X`, since `X` itself can be used for that; for instance the "program" `C = not pre(C,false)` can be translated into

```
C:= true;
loop
   C := not C;
end
```

instead of

```
C:= true; PRE_C:= C;
loop
   C := not PRE_C; PRE_C:= C;
end
```

More generally, if a variable `X` depends on the previous value of `Y`, `X` should be computed before `Y`, if possible. Let us consider an other example, which will be used throughout the remainder of the paper. It has been chosen for the purpose of illustrating the presented techniques rather than for its intuitive meaning (more realistic examples can be found in the bibliography [CPHP87, HCRP91]):

```
program EX
   input i: bool; output n: int; local x, y, z: bool;
   n = if pre(x,true) then 0 else pre(n,0) + 1
   x = if pre(x,true) then false else z
   y = if pre(x,true) then pre(y,true) and i else pre(z,true) or i
   z = if pre(x,true) then pre(z,true)
         else (pre(y,true) and pre(z,true)) or i
end
```

In this program, the only constraint induced by the syntactic dependency relation is that `x` must be computed after `z`. Moreover,

(1) since `n` depends on `pre(x,true)`, it should be computed before `x`;
(2) since `y` depends on `pre(x,true)` and `pre(z,true)`, it should be computed before `x` and `z`;
(3) since `z` depends on `pre(x,true)` and `pre(y,true)`, it should be computed before `x` and `y`;

Now, since there is a contradiction between rules (2) and (3) above, a memory is necessary, for instance for storing the previous value of `z`. We get the following code:

```
n := 0; z := true; y := true; x := false; pre_z := z;
loop
   read(i);
   n := if x then 0 else n + 1;
   z := if x then z else (y and z) or i;
   y := if x then y and i else pre_z or i;
   x := if x then false else pre_z;
   pre_z := z; write(n)
end
```

or, by factorizing the conditional:

```
n:=0; z:=true; y:=true; x:=false; pre_z:=z;
loop
   read(i);
   if x then n:=0; y:=(y and i); x:=false;
   else n:=n+1; z:=(y and z) or i; y:=pre_z or i; x:=pre_z;
   endif;
   pre_z:=z; write(n)
end
```

4 "Data driven" control synthesis

The translation proposed in the previous section appears to often produce a quite inefficient code. As a matter of fact the single loop is a poor control structure. It thus appears that some more sophisticated control structure must be derived, in order to avoid irrelevant tests. For instance, in our example program, one could take advantage of the knowledge that, at a given cycle, `x` takes the value `false` in order to avoid testing the previous value of `x` at the next cycle. The idea is thus to execute a different code at the next cycle, according to the state of the variables at the current cycle.

Moreover such a control structure is suggested by our semantics. According to this semantics, an execution cycle of a program consists of two steps:

1. the current values of the variables are computed
2. the program is rewritten into a new program, which summarizes all the information necessary for executing the next cycle.

A program is therefore an automaton, whose states are the different forms that the program can take as it is rewritten. These forms only differ on the constant values appearing as second operands of the `pre` operators. These values will be called *state variables*.

For instance, consider the program `C = not pre(C, false)`. From the semantic rules, we have (see the example derivation § 2.2):

$$\texttt{C = not pre(C, false)} \xrightarrow{[true/\texttt{C}]} \texttt{C = not pre(C, true)}$$

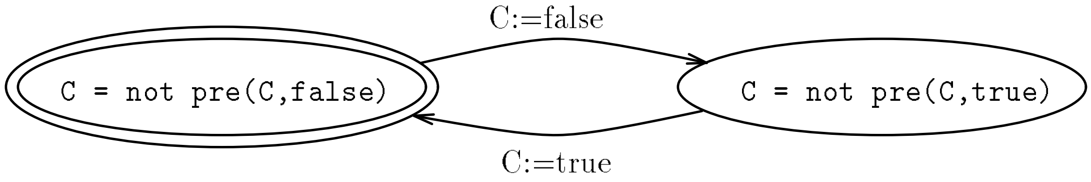

Figure 1: The automaton of the program `C = not pre(C, false)`

and

$$\texttt{C = not pre(C, true)} \stackrel{[false/\texttt{C}]}{\longrightarrow} \texttt{C = not pre(C, false)}$$

So, this program corresponds to an automaton with two states, drawn on Fig. 1

Of course, the automaton may have infinitely many states (since state variables can be numbers), so the idea is to fold it onto a finite automaton which is the wanted control structure. An easy way to perform this finite folding consists of considering only boolean state variables, thus leaving integer computations be performed at runtime: they will be merely placed onto the transitions of the automaton.

So, the control structure is built by simulating the behavior of boolean variables. Let us illustrate this procedure on our example program `EX`. Let eqs_0 be the system of equations defining its boolean variables (this system will be the initial state of the automaton):

```
eqs0 : x = if pre(x,true) then false else z
       y = if pre(x,true) then pre(y,true) and i else pre(z,true) or i
       z = if pre(x,true) then pre(z,true)
              else (pre(y,true) and pre(z,true)) or i
```

In this state, the definition of `n` reduces to `n:=0`. From the semantic rules, we have two possible rewritings, according to the value of the input `i`:

- $eqs_0 \xrightarrow{\sigma_1} eqs_1$, where $\sigma_1 = [true/\texttt{i},\ false/\texttt{x},\ true/\texttt{y},\ true/\texttt{z}]$ and

```
eqs1 : x = if pre(x,false) then false else z
       y = if pre(x,false) then pre(y,true) and i else pre(z,true) or i
       z = if pre(x,false) then pre(z,true)
              else (pre(y,true) and pre(z,true)) or i
```

- $eqs_0 \xrightarrow{\sigma_2} eqs_2$, where $\sigma_2 = [false/\texttt{i},\ false/\texttt{x},\ false/\texttt{y},\ true/\texttt{z}]$ and

```
eqs2 : x = if pre(x,false) then false else z
       y = if pre(x,false) then pre(y,false) and i else pre(z,true) or i
       z = if pre(x,false) then pre(z,true)
              else (pre(y,false) and pre(z,true)) or i
```

Now, two new states (eqs_1 and eqs_2) have been reached. They must be considered in turn:

- In the state corresponding to eqs_1, we have `n:=n+1`, since the previous value of `x` is *false*. Whatever be the value of `i` in σ_3, we get $eqs_1 \xrightarrow{\sigma_3} eqs_0$, provided $\sigma_3(\mathtt{x}) = \sigma_3(\mathtt{y}) = \sigma_3(\mathtt{z}) = true$.

- eqs_2 has two possible rewritings, according to the value of the input `i`:

  $eqs_2 \xrightarrow{\sigma_4} eqs_0$, where $\sigma_4 = [true/\mathtt{i},\ true/\mathtt{x},\ true/\mathtt{y},\ true/\mathtt{z}]$

  $eqs_2 \xrightarrow{\sigma_5} eqs_3$, where $\sigma_5 = [false/\mathtt{i},\ false/\mathtt{x},\ true/\mathtt{y},\ false/\mathtt{z}]$ and

```
eqs3 : x = if pre(x,false) then false else z
       y = if pre(x,false) then pre(y,true) and i else pre(z,false) or i
       z = if pre(x,false) then pre(z,false)
              else (pre(y,true) and pre(z,false)) or i
```

Considering all the states in turn, we get:

- eqs_3 has two possible rewritings, according to the value of the input `i`:

  $eqs_3 \xrightarrow{\sigma_4} eqs_0$

  $eqs_3 \xrightarrow{\sigma_6} eqs_4$, where $\sigma_6 = [false/\mathtt{i},\ false/\mathtt{x},\ false/\mathtt{y},\ false/\mathtt{z}]$ and

```
eqs4 : x = if pre(x,false) then false else z
       y = if pre(x,false) then pre(y,false) and i else pre(z,false) or i
       z = if pre(x,false) then pre(z,false)
              else (pre(y,false) and pre(z,false)) or i
```

- eqs_4 has two possible rewritings, according to the value of the input `i`:

  $eqs_4 \xrightarrow{\sigma_4} eqs_0 \qquad eqs_4 \xrightarrow{\sigma_6} eqs_4$

All the reached states have been processed. In all states eqs_3, eqs_4, eqs_5, since the previous value of `x` is false, we get `n:=n+1`. The resulting automaton is drawn in Fig 2. A possible code would be the following:

```
EQS0 :  n:=0 ; write(n); read(i); if i then goto EQS1 else goto EQS2;
EQS1 :  n:=n+1 ; write(n); read(i); goto EQS0;
EQS2 :  n:=n+1 ; write(n); read(i); if i then goto EQS0 else goto EQS3;
EQS3 :  n:=n+1 ; write(n); read(i); if i then goto EQS0 else goto EQS4;
EQS4 :  n:=n+1 ; write(n); read(i); if i then goto EQS0 else goto EQS4;
```

Such a method was first introduced for the compilation of Esterel [BG92] and is used by the Lustre-V2 compiler [CPHP87, Pla88]. Efficient algorithms exist for it, since it is similar to the method for building an accepting automaton for a regular expression [Brz64, ASU86, BS87]. The resulting code appears to be minimal, in some sense, with respect to the execution time.

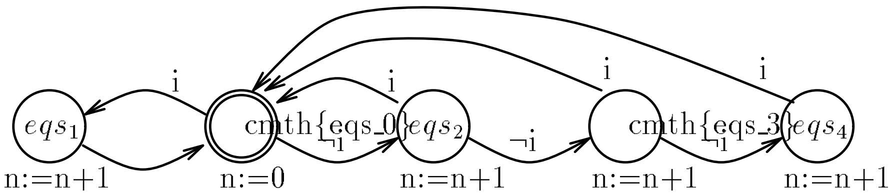

Figure 2: The control automaton of the program `EX`

5 "Demand driven" control synthesis

Experience with ESTEREL shows that the resulting automaton is generally small (while theoretically exponential), and close to the minimal one (with respect to bisimulation, the usual minimization criterion over automata). However, it is not the case with a declarative language. As a matter of fact, the automaton built from our example is not minimal (eqs_2, eqs_3 and eqs_4 are equivalent states).

The problem is that the declarative style encourages the definition of variables without care of their effective use: variables are defined at each cycle, but if they are only intermittently used, the automaton obtained by the above procedure may contain many equivalent states, which differ only in the values of state variables which affect neither the current outputs nor the future behavior of the program. Notice that the size of the automaton does not affect the execution time of the code (as far as this code does not exceed the memory size) but a tremendous extension of the code size is clearly unrealistic. A possible solution would be to apply a standard minimization algorithm [PT87, Fer90], which still supposes there is enough memory to store the automaton. A better solution consists of integrating the minimization into the generation process. So, we have designed a algorithm [BFH90] for generating directly the minimal automaton, by computing state variables according to a "demand-driven" strategy: A variable is computed in a state if and only if its value is necessary for computing either the output or the next state. In this section, we illustrate the use of this algorithm on our toy language.

The basic idea is the following: We want to avoid distinguishing between states, when they differ on state variables which are used neither in the current output nor in the future behavior of the program. Hence, we shall deal with classes of equivalent states, characterized by boolean formulas. The method is inspired from the standard minimization of automata, whose principles are the following:

- Two states are considered equivalent as long as they haven't been shown to be different;
- Two states are different if and only if, in response to the same inputs, either they produce different outputs, or they lead to states which have been shown to be different.

In a similar way, our demand-driven generation of the automaton will progressively refine a partition of the set of states. The initial partition consists of one class, i.e., all the states are considered equivalent. Processing a class C consists of generating the code for its outputs and computing its successor classes. Whenever this computation involves some unknown state information ι, the class C is split into two subclasses: the subset C_0 of states for which ι is false, and the subset C_1 of those for which ι is true. Then, predecessor classes of C are asked for the value of ι, and requested to choose their successor among C_0 and C_1.

In order to make clear the state information needed for computing an expression of the program, we shall put the expression into a normal form:

Definitions: An *instantaneous expression* is an expression which contains neither a `pre` operator applied to a boolean operand, nor any boolean variable which is not an input. An expression E is in *normal form* if and only if either it is an instantaneous expression, or it is of the form:

```
if pre(E₁,v₁) then F₁
else if pre(E₂,v₂) then F₂
else ...
else if pre(Eₙ₋₁,vₙ₋₁) then Fₙ₋₁
else Fₙ
```

where all the F_i are syntacticly different instantaneous expressions. It can be easily shown that any expression of the language can be put into such a form.

Now, let us apply the demand-driven generation to our example program `EX`: We shall consider the program written as follows, where boolean state variables are noted v_x, v_y, v_z:

```
program EX
   input i:  bool; output n:  int; local x, y, z:  bool;
   n = if pre(x,vx) then 0 else pre(n,0) + 1
   x = if pre(x,vx) then false else z
   y = if pre(x,vx) then pre(y,vy) and i else pre(z,vz) or i
   z = if pre(x,vx) then pre(z,vz)
         else (pre(y,vy) and pre(z,vz)) or i
end
```

State classes will be characterized by predicates on v_x, v_y, v_z. We start with one class C, characterized by the formula *true*, in which we try to compute the output `n`. The expression defining `n`, "`if pre(x,`v_x`) then 0 else pre(n,0) + 1`", is already in normal form, which shows that the code for `n` depends on the state information v_x. So C is split into two subclasses: C_0, where the value of v_x is false, and C_1 where the value of v_x is true. So, in C_0 (resp. C_1), `pre(x,`v_x`)` is false (resp. true) and the code for the output is `n:=n+1` (resp. `n:=0`). Since in the initial form of the program, v_x is written `true`, the initial state belongs to C_1, which is (until now) the only clearly accessible class.

Let us process C_1: The code for the output is `n:=0`. Let us compute the next class, i.e, answer the question: Will v_x be true or false at the next cycle? The answer depends on the value of `x` in C_1, so, we have to compute `x`. The expression defining `x`, `if pre(x,`v_x`) then false else z`, must be put in normal form. We get:

```
x = if pre(x,vx) then false else if pre(y and z,vy and vz) then true else i
```

In C_1, where v_x is true, it evaluates to false. So the next class is C_0, which is thus accessible, and must be processed.

In C_0, the code for the output is `n:=n+1`. For computing the next class, we have also to compute `x`, which evaluates to

```
if pre(y and z,vy and vz) then true else i
```

So, the next class depends on the state information (v_y `and` v_z). So, C_0 is split into

- C_{00} where both v_x and (v_y `and` v_z) are false;
- C_{01} where v_x is false and where (v_y `and` v_z) is true.

The predecessor class of C_0, C_1 is requested to choose between these two subclasses, that is to compute the value of `(y and z)`, the normal form of which is:

```
if pre((x and y and z) or (not x and not(y and z)),
            (vx and vy and vz) or (not vx and not (vy and vz)))
then i
else if pre(x and not(y and z),vx and not(vy and vz))
then false else true
```

In C_1, this expression reduces to

```
if pre(y and z, vy and vz) then i else false
```

So, the choice of the successor class of C_1 among C_{00} and C_{01} depends on the state information (v_y `and` v_z), which is unknown in C_1. So, C_1 is split into:

- C_{10}, where v_x is true and (v_y `and` v_z) is false;
- C_{11}, where both v_x and (v_y `and` v_z) are true.

The initial state belongs to C_{11}, which is the only accessible class, since until now, C_1 doesn't have any predecessor.

From the definition of C_{11}, (v_y `and` v_z) is true in C_{11}, so the value of `(y and z)` is `i`, and the choice of the successor class of C_{11} depends on the value of `i`: if `i` is false, the next class will be C_{00}, otherwise it will be C_{01}. As a consequence, both C_{00} and C_{01} are accessible classes, and must be processed.

In these classes, we already know that the code for the output is `n:=n+1` (since we consider subclasses of C_0), so only the successor class must be chosen, among C_{00}, C_{01}, C_{10} and C_{11}, according to the values of `x` and `(y and z)`:

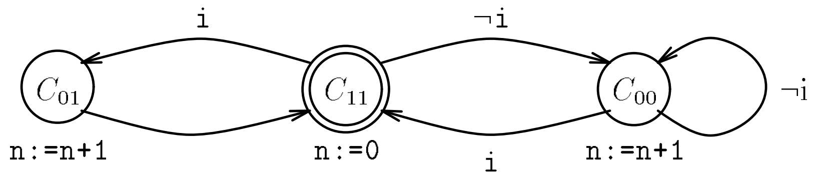

Figure 3: Minimal automaton of `EX`

- In C_{00}, both v_x and (v_y **and** v_z) are false, so both `x` and `(y and z)` evaluate to `i`. So if `i` is false the successor class is C_{00}, otherwise it is C_{11}.
- In C_{01}, v_x is false and (v_y **and** v_z) is true. From their normal forms, both `x` and `(y and z)` evaluate to true. So the successor class is C_{11}.

All the accessible classes have been processed, so the generation is complete, and we are left with the automaton of Fig. 3. Compared with the "data-driven" strategy applied in section 4, notice that the three equivalent states eqs_2, eqs_3, eqs_4 have not been distinguished: they all belong to the class C_{00}. Notice also that our algorithm only deals with accessible classes: The class C_{01}, which is not accessible, has not been considered for splitting. This makes the distinction with standard minimization algorithms.

6 Implementation and experimentation

As mentioned before, all the methods presented here have been implemented in the LUSTRE compilers. In this section, we shall discuss the experience gained from this implementation, and particularly from the LUSTRE-V3 compiler. This compiler is written in C++, and translates a LUSTRE program into a sequential C program.

Control synthesis: The synthesis of the control structure presented in section 4 improves the code performances of a rate ranging from 10% to 50%, according to the proportion of boolean computations involved in the source program. For instance, a digital watch written in LUSTRE and compiled into an automaton gives rise to 41 states, and runs about 20% faster than the same program compiled as a single loop. Practically, its stopwatch counts the 1/100 second (on a SUN3/60) in the former case, but not in the later. For a highly boolean program, like the example `EX` considered in the paper, the improvement in speed is about 50%. The counterpart of the improvement in speed is the code size, which is often large. However, the code is produced in a particular form, where actions are stored in table, so as to avoid copying their code in each state of the automaton. This way of producing the code [PS87] is common with the ESTEREL compiler. Moreover, their are several options for choosing state variables, so as to let the user manage the compromise between the speed and the code size.

"Demand driven" synthesis: The interest of minimizing the automaton during its generation appeared from experience. As a matter of fact, in many cases, the "data driven" method described in section 4 produces a lot of equivalent states, or cannot succeed in producing the automaton because of these irrelevant states, and over all, because of irrelevant transitions between equivalent states. One can be easily convinced that the number of transitions grows as the square of the number of equivalent states. This explains why the demand-driven algorithm runs generally faster, since it deals only with the minimal number of states and transitions. The demand-driven algorithm involves symbolic computation of boolean formulas. An efficient decision procedure has been implemented by means of Bryant's "Binary Decision Diagrams" [Bry86].

The following tables compare the generation times (on SUN 4) and the results of four compiling strategies, applied to our example `EX` and to the digital watch program:

- the single loop generation
- the data-driven generation alone
- the data-driven generation followed by a minimization by ALDEBARAN [Fer90], an automaton minimizer based on the Paige-Tarjan algorithm [PT87]
- the demand-driven generation

Times are given in second, and code sizes in Kbytes.

Program `EX`	single loop	data-driven	data-driven + minimization	demand-driven
Compilation time	0.5	0.5	1.2	0.3
Number of states	1	5	3	3
Number of transitions	1	11	5	5
Size of the generated code	1.56	0.89	0.85	0.85
Average reaction time	29μs	15μs	15μs	15μs

Program `WATCH`	single loop	data-driven	data-driven + minimization	demand-driven
Compilation time	1.3	8.1	41.5	6.8
Number of states	1	81	41	41
Number of transitions	1	1163	474	342
Size of the generated code	7.88	30.94	19.15	16.51
Average reaction time	820μs	590μs	590μs	578μs

Aknowledgements: We are indebted to Gérard Berry for the idea of synthesizing the control structure of a program as a finite automaton. John Plaice wrote the compiler LUSTRE-*V2 which implements the data-driven automaton generation. Many ideas about the ordering of computations along the transitions of the automaton are also due to him. Christian Berthet*

and Jean-Christophe Madre taught us how to implement a boolean decision procedure by means of "binary decision diagrams", without which the demand-driven algorithm would not work so well.

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers : Principles, Techniques and Tools.* Addison-Wesley, 1986.

[AW85] E. A. Ashcroft and W. W. Wadge. LUCID*, the data-flow programming language.* Academic Press, 1985.

[BCHP86] J-L. Bergerand, P. Caspi, N. Halbwachs, and J. Plaice. Automatic control systems programming using a real-time declarative language. In *IFAC/IFIP Symp. 'SOCOCO 86, Graz*, May 1986.

[BFH90] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal model generation. In R. Kurshan, editor, *International Workshop on Computer Aided Verification, Rutgers*, June 1990.

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

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.

[Brz64] J. A. Brzozowski. Derivative of regular expressions. *JACM*, 11(4), 1964.

[BS87] G. Berry and R. Sethi. From regular expressions to deterministic automata. *TCS*, 25(1), 1987.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, Munchen, January 1987.

[Fer90] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3), May 1990.

[Gra82] J. R. Mc Graw. The VAL language: Description and analysis. *ACM TOPLAS*, 4(1), January 1982.

[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.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74*. North Holland, 1974.

[Pla88] J. A. Plaice. Sémantique et compilation de LUSTRE, un langage déclaratif synchrone. Thesis, Institut National Polytechnique de Grenoble, Grenoble, France, 1988.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Lecture notes, Aarhus University, 1981.

[PP83] N.S. Prywes and A. Pnueli. Compilation of nonprocedural specifications into computer programs. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.

[PS87] J. A. Plaice and J-B. Saint. The LUSTRE-ESTEREL portable format. Unpublished report, INRIA, Sophia Antipolis, 1987.

[PT87] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6), 1987.