STATEFLOW TO LUSTRE

Defining and translating a safe subset of Simulink/Stateflow into Lustre.
Verimag verion Vs Automaton version.

Hamza BOURBOUH
ISAE / ONERA / NAZA Ames

Plan

1. Abstract

2. A safe subset of Stateflow

2.1 A short description of Stateflow

2.2 The interpretation algorithm

2.3 Semantic issues with Stateflow

3. Simple conditions identifying a “safe” subset of Stateflow

4. Lustre and automaton

5. Translation into Lustre

5.1 Encoding of states Verimag version

5.2 Translation into Automaton : First Version.

1. Abstract

STATEFLOW is problematical for synchronous languages because of its unbounded behavior so we propose analysis techniques to define a subset of STATE-FLOW for which we can define a synchronous semantics. We go further and define a “safe” subset of STATEFLOW which elides features which are potential sources of errors in STATEFLOW designs.

2. A safe subset of Stateflow

- ✓ **A short description of Stateflow**
- ✓ **The interpretation algorithm**
- ✓ **Semantical issues with Stateflow**

2.1 A short description of Stateflow

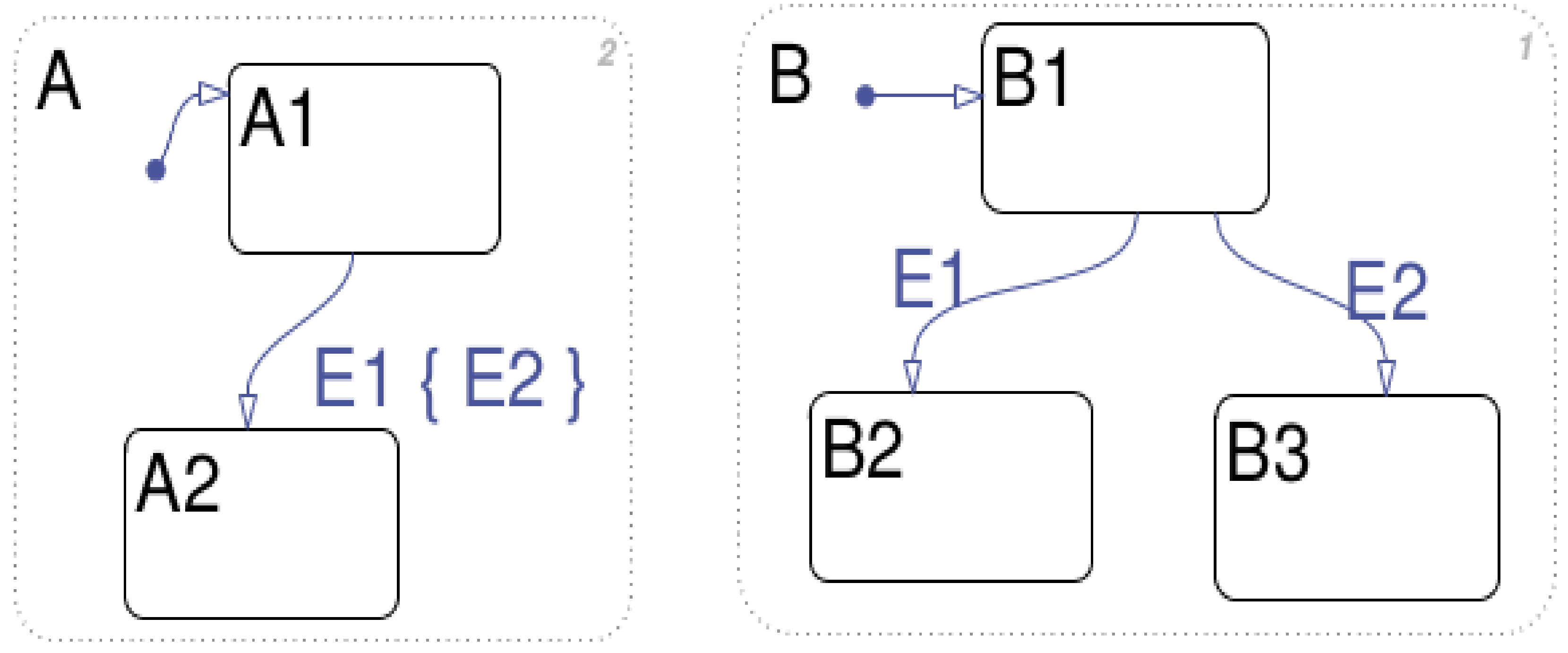


Figure 3: Example of non-confluence

states can be refined into either exclusive (OR) states connected with transitions or parallel (AND) states, which are not

2.1 A short description of Stateflow

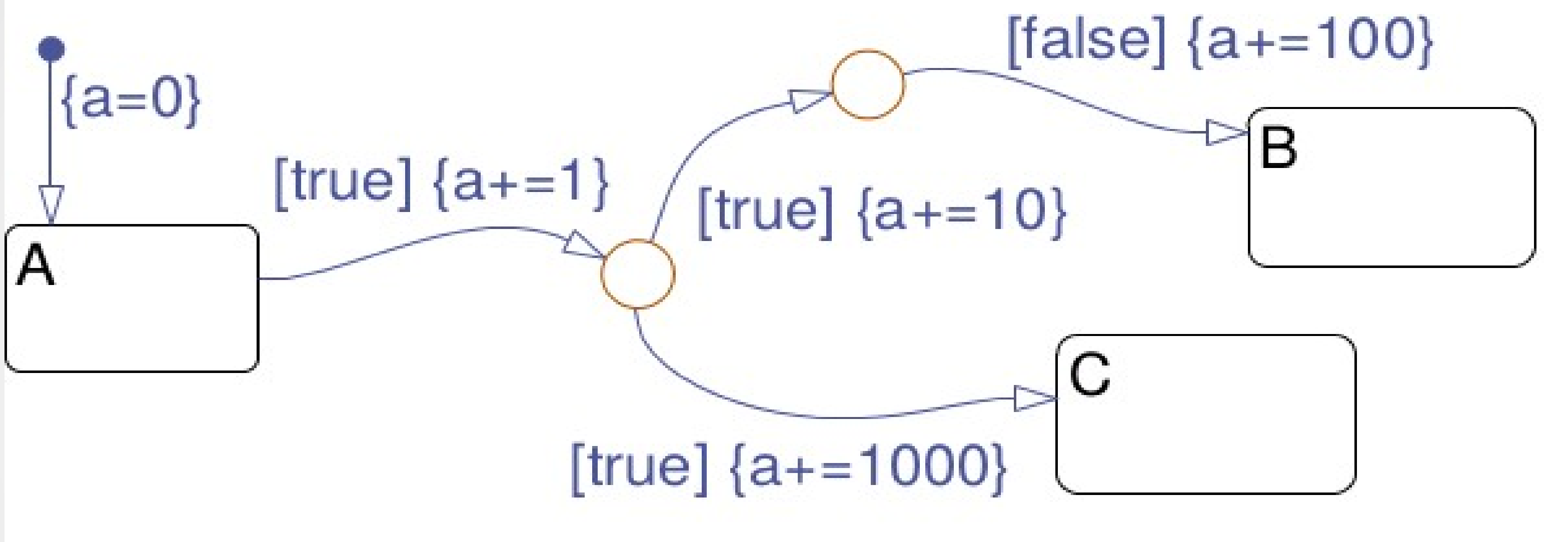


Figure 2: Example of backtracking

A transition can be a complex (possibly cyclic) flow graph made of segments joining connective junctions.

2.1 A short description of Stateflow

Each segment can bear a complex label with the following syntax (all fields are optional):

E[C]{Ac }/At

where E is an event, C is the condition (i.e., guard), Ac is the condition action and At is the transition action. Ac and At are written in the action language of STATEFLOW , which contains **assignments, emissions of events**, and so on. Actions written in the action language can also annotate states. **A state can have an entry action, a during action, an exit action and on event E actions, where E is an event.**

2.2 The interpretation algorithm

Search for active states: in order to impose determinism upon Stateflow semantics, the search for active states is performed from **top to bottom and from left to right**,

Search for valid transitions: transitions are searched according to the **12 o’clock rule.** when the transition is multi-segment, the condition actions of each segment are executed while searching and traversing the transition graph.

Execute a valid transition: execute the exit action of the source state, set the source state to inactive, execute the transition actions of the transition path, set the destination state to active and finally execute the entry action of the destination state.

Idling: when an active state has no valid output transitions an active state performs its during action and the state remains active.

Termination: occurs when there are no active states.

2.2 The interpretation algorithm

It should be emphasized that **each of the executions runs to completion** and this makes the behavior of the overall algorithm very complex. In particular, when any of the actions consists of **broadcasting an event**, the interpretation algorithm for that event is also run to completion before execution proceeds. This means that the interpretation algorithm is **recursive** and uses a stack. However, as we will see, the stack does not store the full state, which leads to problems of side effect (**Backtracking without “undo” example**). Also, without care, the stack may overflow (**Non-termination and stack overflow example**).

2.3 Semantic issues with Stateflow

Non-termination and stack overflow

Backtracking without "undo"

Dependence of semantics on graphical layout

early return logic

Non-termination and stack overflow

When the default state A is entered event E is **emitted** in the **entry action** of A. E results in a **recursive** call of the **interpretation algorithm** and since A is active its outgoing transition is tested. Since the current event E matches the transition event (and because of the absence of condition) the condition action is executed, emitting E again. This results in a new call of the **interpretation algorithm** which repeats the same sequence of steps until stack overflow.

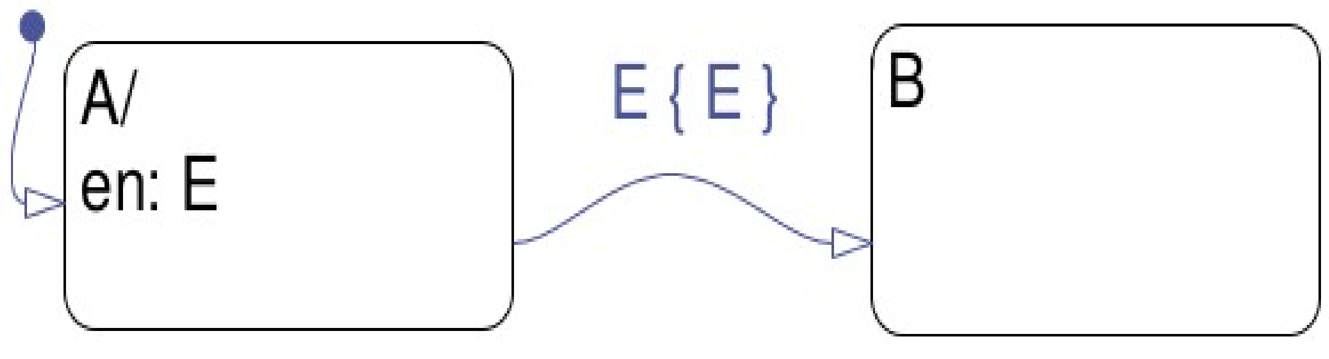


Figure 1: Stack overflow

Backtracking without “undo”

An example of such a behavior is generated by the model shown in Figure 2. The final value of variable a when state C is entered will be 1011 and not 1001 as might be expected. This is because when the segment with condition “false” is reached, the algorithm backtracks without “**undoing**” the action “a+=10”.

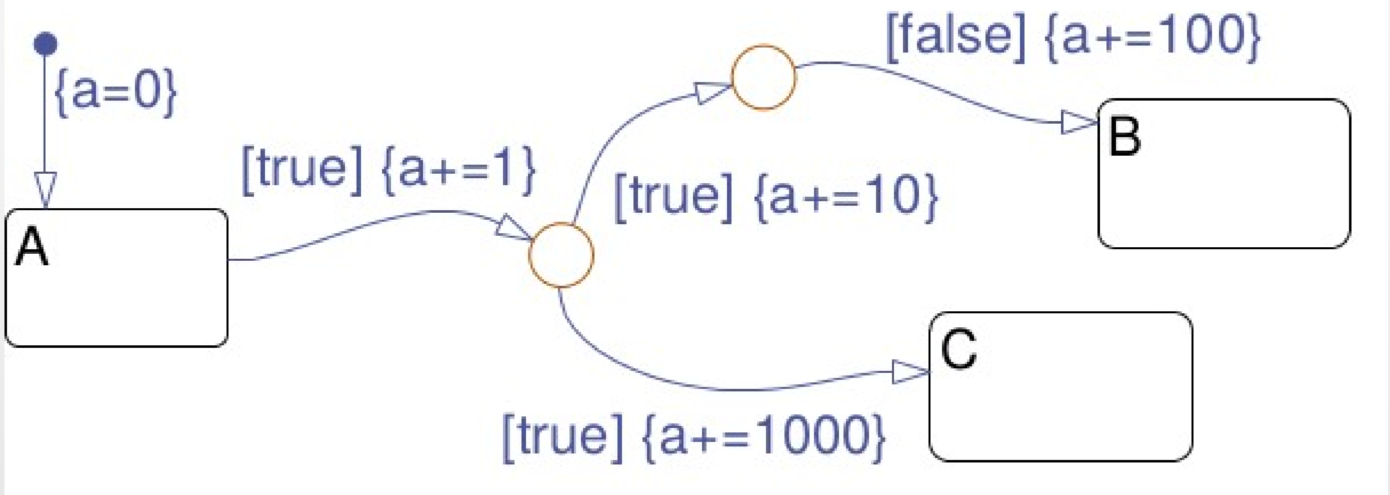


Figure 2: Example of backtracking

Dependence of semantics on graphical layout

A and B are **parallel states**. When **event** E1 arrives, if A is explored first, then E2 will be **emitted** and the final global state will be (A2, B3). But if B is explored first then the final global state will be (A2, B2). Thus, parallel states in STATEFLOW do not enjoy the property of **confluence**.

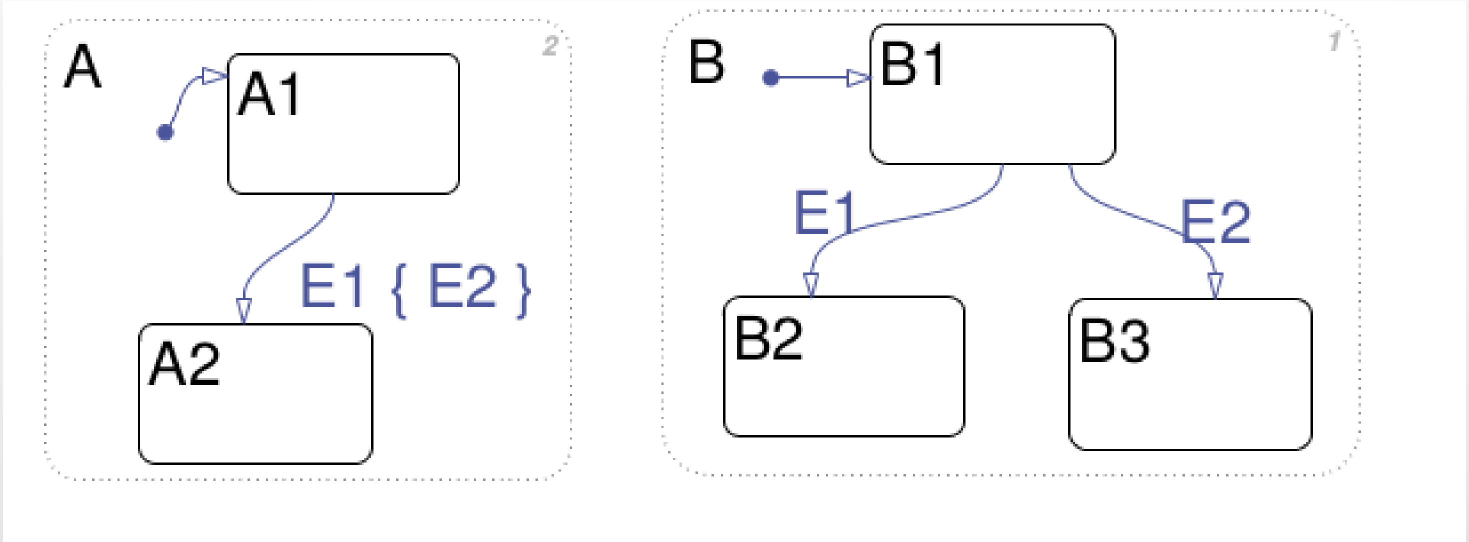


Figure 3: Example of non-confluence

early return logic

When **event** E is **emitted**, the **interpretation algorithm** is called **recursively**. Parent state A is **active**, thus, its outgoing transition is explored and, since **event** E is present, the **transition** is taken. This makes A **inactive**, and B **active**. When the stack is popped and execution of the previous instance of the interpretation algorithm resumes, **state** A1 is not **active** anymore, since its parent is no longer **active**.

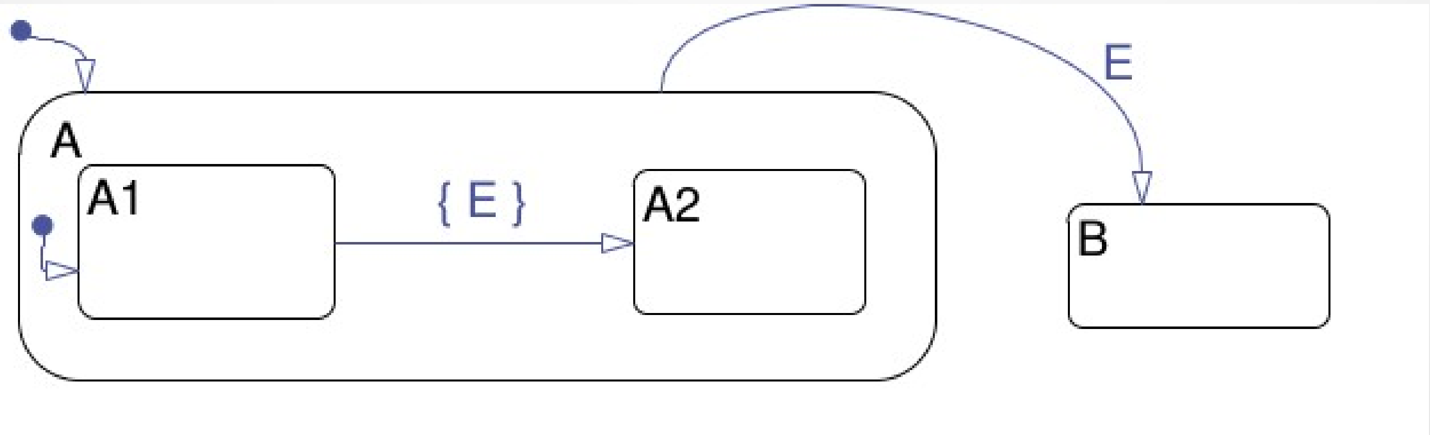


Figure 4: "Early return logic" problem

3 Simple conditions identifying a “safe” subset of Stateflow

In this section we present a sufficient number of simple conditions for avoiding error-prone models such as those discussed previously.

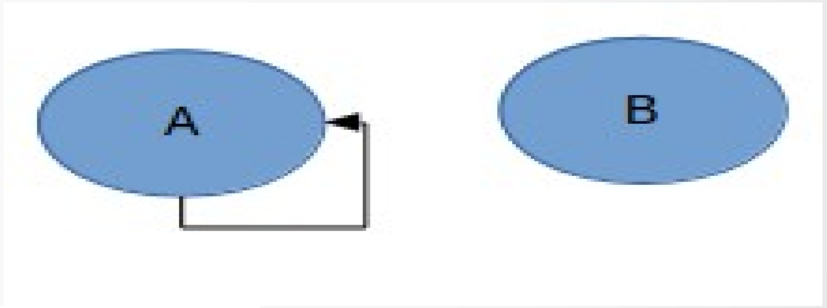
- **Absence of multi-segment loops:** If no graph of junctions and transition segments contains a loop (a condition which can easily be checked statically) then the model will not suffer from **non-termination** problems referred previously.
- Notation : An **event** E is said to be **triggering a state** s if the **state** has a “**on event** E: A” **action** or an **outgoing transition** which can be **triggered** by E . E is said to be **emitted** in s if it appears in the **entry**, **during**, **exit** or **on-event action** of s, or in the **condition** or **transition action** of one of the **outgoing transitions** of s.

Acyclicity in the graph of triggering and emitted events

We draw a graph where nodes are the states and transitions between two nodes (v, v') exist iff state v can emit event E which can then trigger state v' , but only if v and v' can be active at the same time. If the graph above has no directed cycle then the model will not suffer from stack overflow problems.

Example fig1 :

E is emitted in A and triggering A.



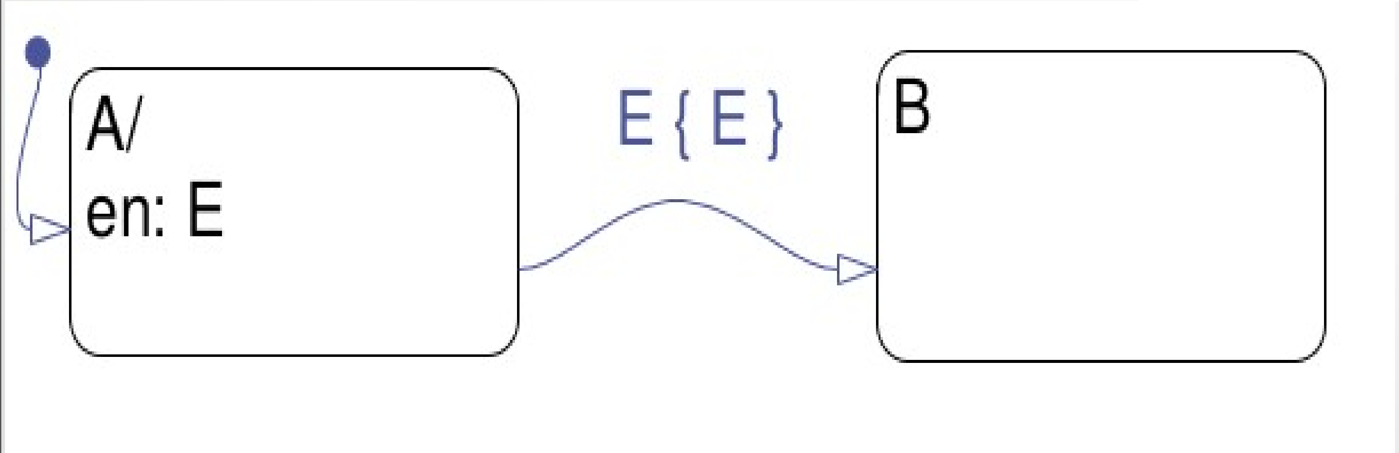


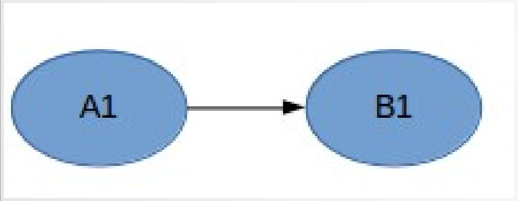
Figure 1: Stack overflow

Absence of assignments in intermediate segments:

- In order to avoid side effects due to lack of “undo”, we can simply check that all variable assignments in a multi-segment transition appear either in **transition actions** (which are executed only once a destination state has been reached) or in **the condition action of the last segment** (whose destination is a state and not a junction). This ensures that even in case the algorithm backtracks, no variable has been modified. An alternative is to avoid backtracking altogether.

Checks for confluence: (see fig3)

- A simple solution is to check that in the aforementioned graph of triggering and emitted events, there is no edge v→v' such that v belongs to A and v' to B or vice-versa.
- In this example E2 is emitted in A1 and triggering B1.



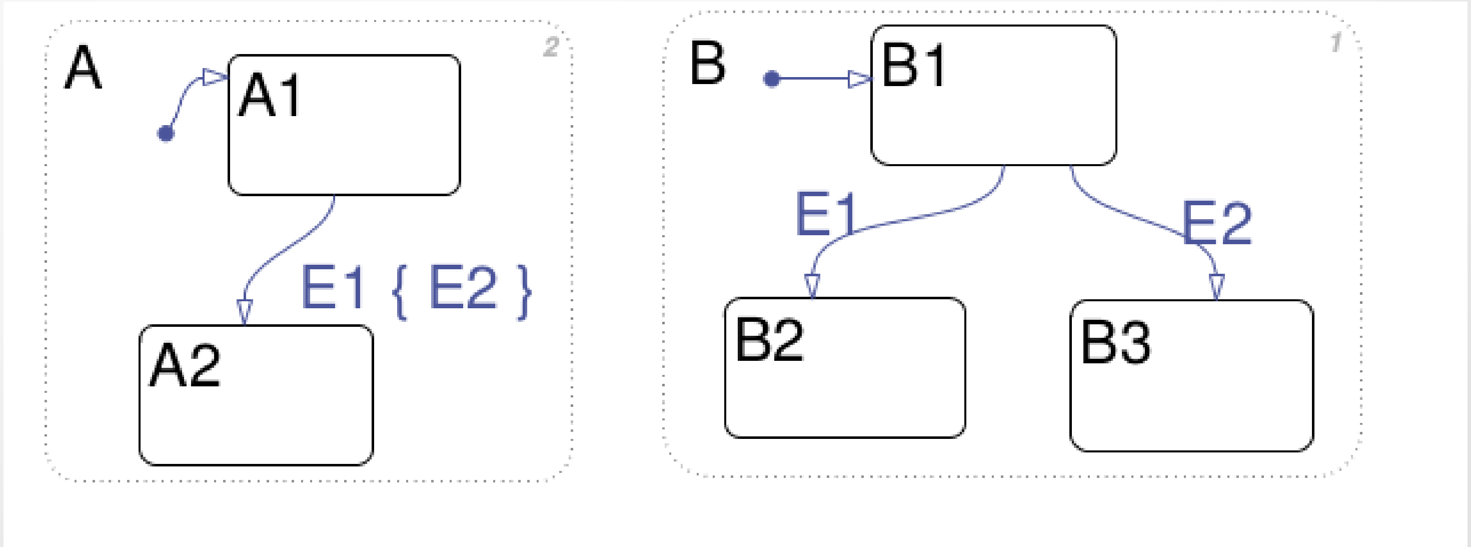


Figure 3: Example of non-confluence

Checks for “early return logic”:(fig4)

- To ensure that our model is free of “early return logic” problems, we can check that for every state s and each of its outgoing transitions having a triggering event, this event is not emitted somewhere in s. In this example E is triggering A1 (because the event field is empty) and is emitted in A1 (because it appears in the condition action).

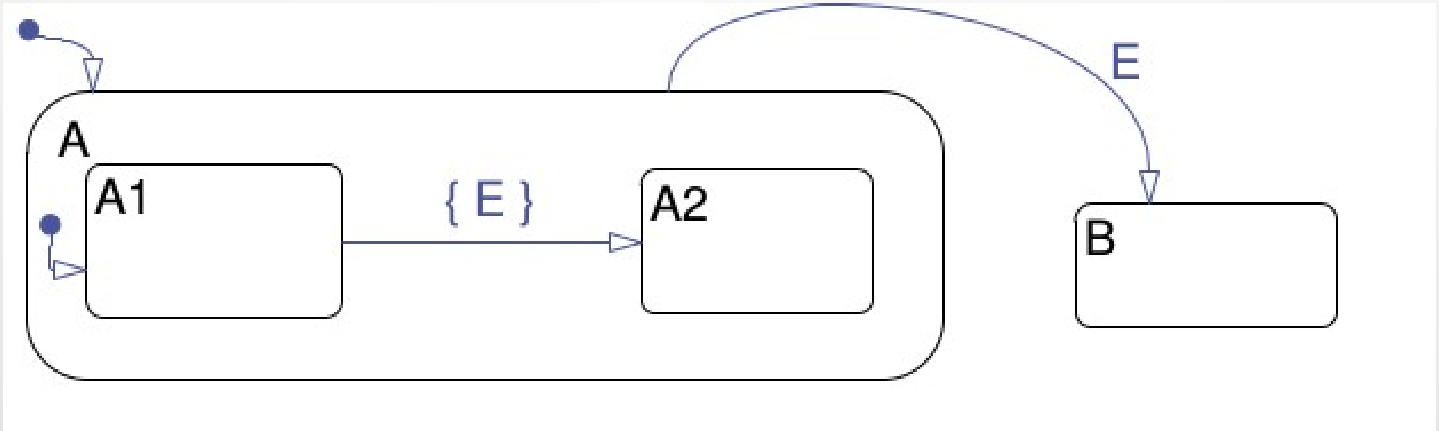


Figure 4: “Early return logic” problem

4. Lustre and automaton

- Syntaxe :

[declaration of types and external functions]

node name (declaration of input flows)
returns (declaration of output flows)
    [**var** declaration of local flows]
let
    [assertions]
    system of equations defining once each local flow and output depending on them and the inputs.
tel

[other nodes]

4. Lustre and automaton

- Operator pre :

  Memorization of the precedent value of a flow or a set of flows :

  let X be the flow (X0, X1,..., Xn,...)

  Then pre(X) is the flow (nil,X0, X1,..., Xn,...)

- Operator → :

  Initialization of a flow.

  Let X be the flow (X0, X1,...,Xn,...) and Y the flow (Y0, Y1,..., Yn, …)

  then Y → X is the flow (Y0,X1,...,Xn,...).

4. Lustre and automaton

- Operator when :

  Let X be a flow and B a Boolean flow (assimilated to a clock) of same clock.

  The equation **Y = X when B**

  defines a flow Y, of same type than X, and of clock B

  - Y is present When B is true
  - Y is absent when B is false or when B and X are absent.

4. Lustre and automaton

- Operator current :

  Let X be a flow and B a Boolean flow (assimilated to a clock) of same clock.

  The equation **Y = X current B**

  defines a flow Y, of same type than X, and of clock the clock of B

  - Y is present When B is true
  - When Y is present, Y is equal to X if X is present and to the previous value of X otherwise.

4. Lustre and automaton

- Recapitulation :

B	False	true	false	true	false	false	true	true
X	x0	x1	x2	x3	x4	x5	x6	x7
Y	y0	y	y2	y3	y4	y5	y6	y7
pre(X)	nil	x0	x1	x2	x3	x4	x5	x6
Y → pre(X)	y0	x0	x1	x2	x3	x4	x5	x6
Z = X when B		x1		x3			x6	x7
T=current(Z)	nil	x1	x1	x3	x3	x3	x6	x7
pre(Z)		nil		x1			x3	x6
0 → pre(Z)		0		x1				x6

4. Lustre and automaton

- Automaton :

```
automaton ABC
   initial state A
   unless if e resume B
   let
       z = 0→(pre(Z)+1) ;
   tel
   until  if (not e) resume C

   state B
   unless if e resume C
   let
       z = 0→(pre(Z)+2) ;
   tel

   state C
   let
       z = 0→(pre(Z)+3) ;
   tel
   until if e resume A
returns z ;
```

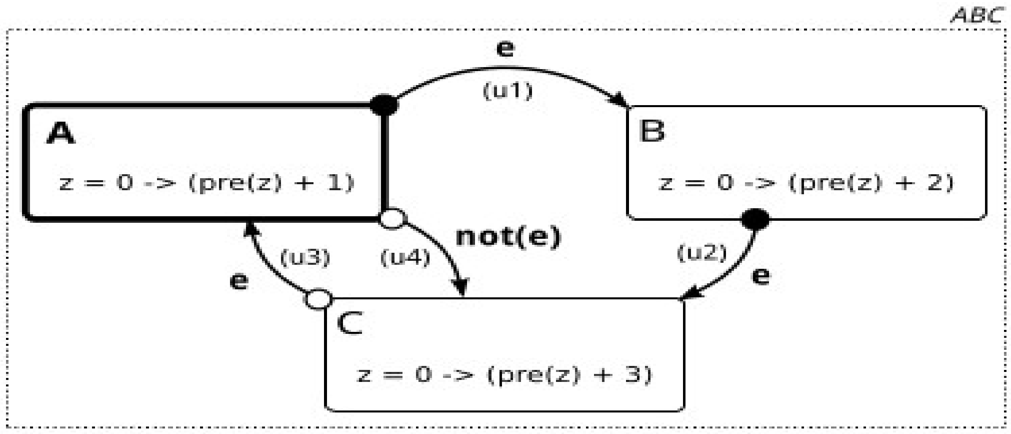


FIGURE 5.2: Automate ABC

4. Lustre and automaton

- Automaton :

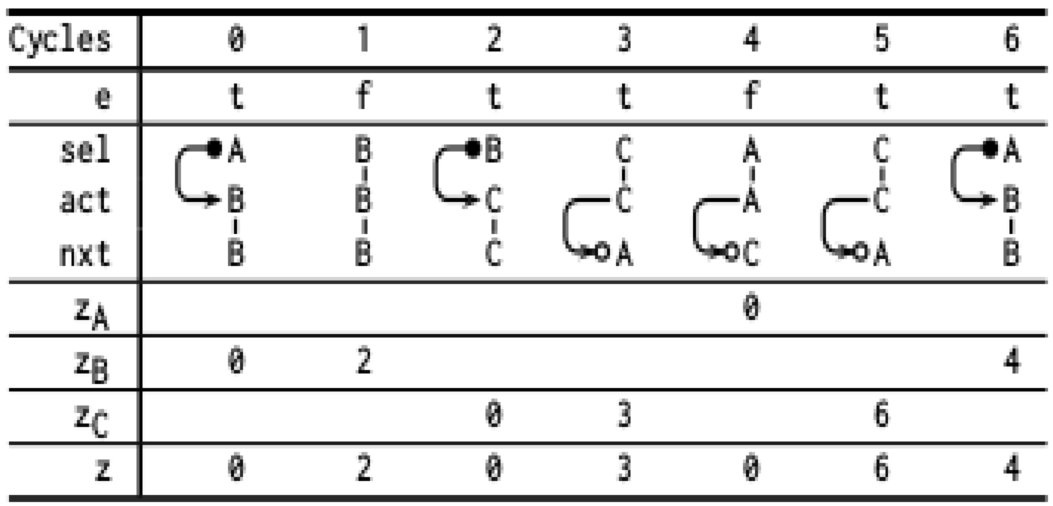
Unless E : if E then go directly to destination state and execute It;

else execute the source state (in this case next state is the Current state state).

Until E : execute source state ; if E then next state will be the Destination of the transition;

else next state is the current state.

Sel : selected state. **Act** : actual state (where we will execute equation). **Nxt** : selected state for next clock.



Cycles	0	1	2	3	4	5	6
e	t	f	t	t	f	t	t
sel	A	B	B	C	A	C	A
act	B	B	C	C	A	C	B
nxt	B	B	C	A	C	A	B
z_A					0		
z_B	0	2					4
z_C			0	3		6	
z	0	2	0	3	0	6	4

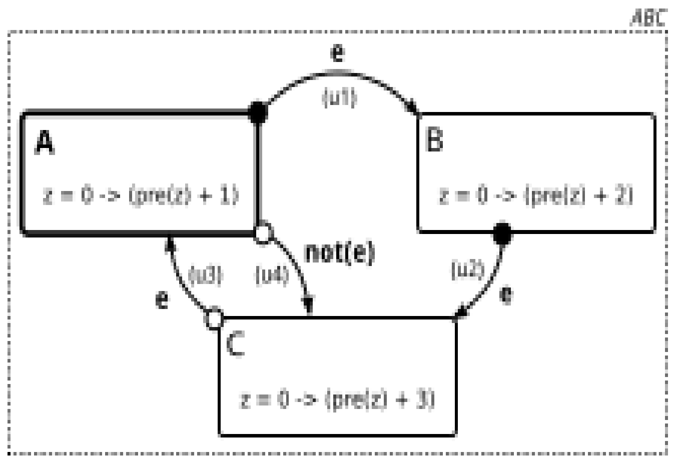


FIGURE 5.3: Trace d'exécution de l'automate *ABC*.

5. Translation into Lustre

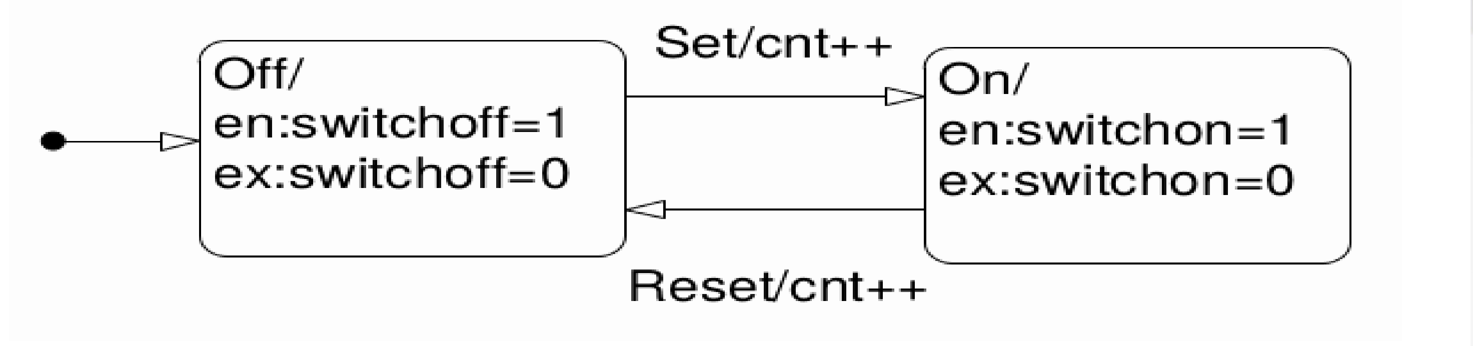
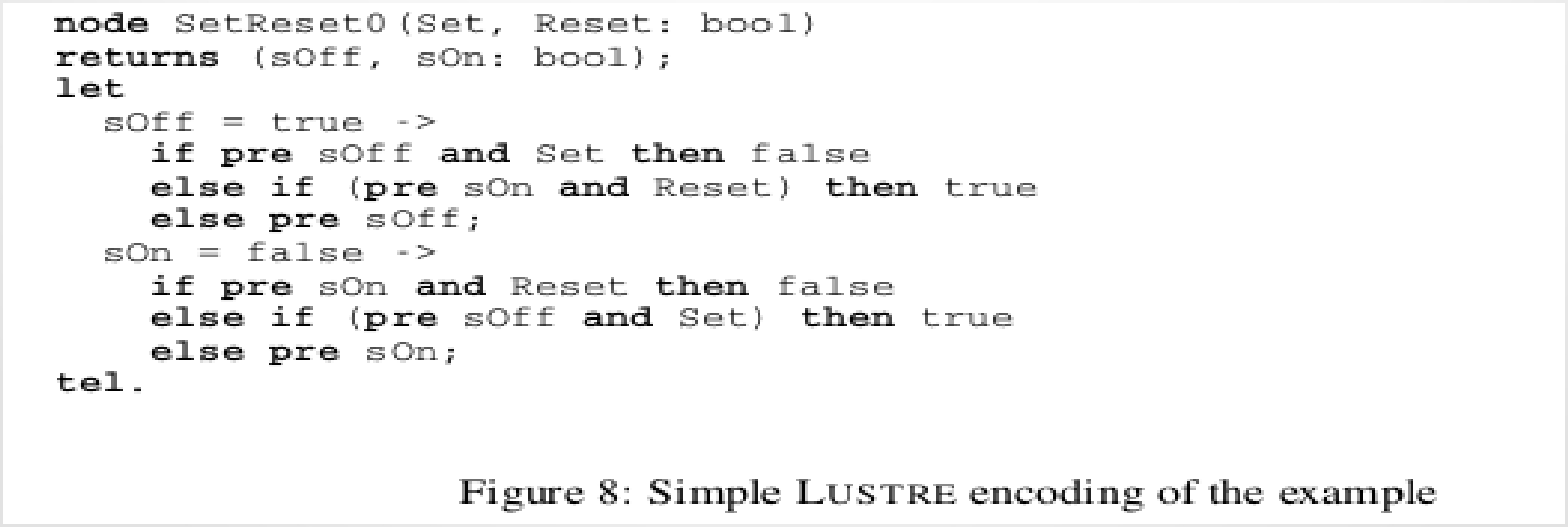


Figure 6: A simple STATEFLOW chart



```
node SetReset0(Set, Reset: bool)
returns (sOff, sOn: bool);
let
  sOff = true ->
    if pre sOff and Set then false
    else if (pre sOn and Reset) then true
    else pre sOff;
  sOn = false ->
    if pre sOn and Reset then false
    else if (pre sOff and Set) then true
    else pre sOn;
tel.
```

Figure 8: Simple LUSTRE encoding of the example

5.1 Encoding of states

- This code is semantically correct for a system consisting only of states but it is difficult to incorporate the imperative actions attached to both states and transitions in STATEFLOW . For example, if the above code had included the entry actions in the states then all the values referenced by the action code would have to be updated in each branch of the if-then tree. So the number of values being updated can become large and if more than one state updates the same value then causality loops could arise.

5.1 Encoding of states

- Inspecting the code in Figure 8 the state update equation for each state consists of:

  - An initialization value computed from default transitions (true for sOff),
  - a value for each outgoing transition (Set for sOff),
  - an exit clause ((pre sOff and Set) for sOff),
  - an entry clause ((pre sOn and Reset) for sOff) and
  - a no-change value (pre sOff).

  Explicitly separating these components allows us to insert the action code at the correct point in the computation of a reaction.

5.1 Encoding of states Verimag version

- the code has been split into several sections.

  • **Initial values.** These are the initial values for all variables, false for states and the initial value from the data dictionary for STATEFLOW variables.

  • **Transition validity.** In this section the values for the transitions are computed. For convenience in the translator these are actually calls to predefined nodes generated in advance from the transitions' events and actions. Note that the test for the activity of the source state is included in the transition's validity test.

  • **State exits.** Any states which are true and have a valid outgoing transition are set to false.

  • **Exit actions.** The code for any exiting state's exit actions is computed. This section also includes during actions for states which remain active and on actions also for active states.

  • **Transition actions**. The code for the transition actions is executed. Note that the exiting state's value is false while this occurs.

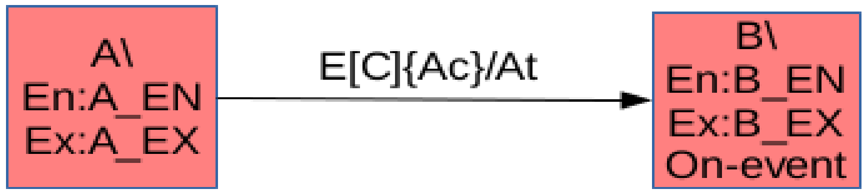
  • **State entries.** Any states which are false and have a valid incoming transition are set to true.

  • **Entry actions.** Entering states action code is executed with the state's variable now true.

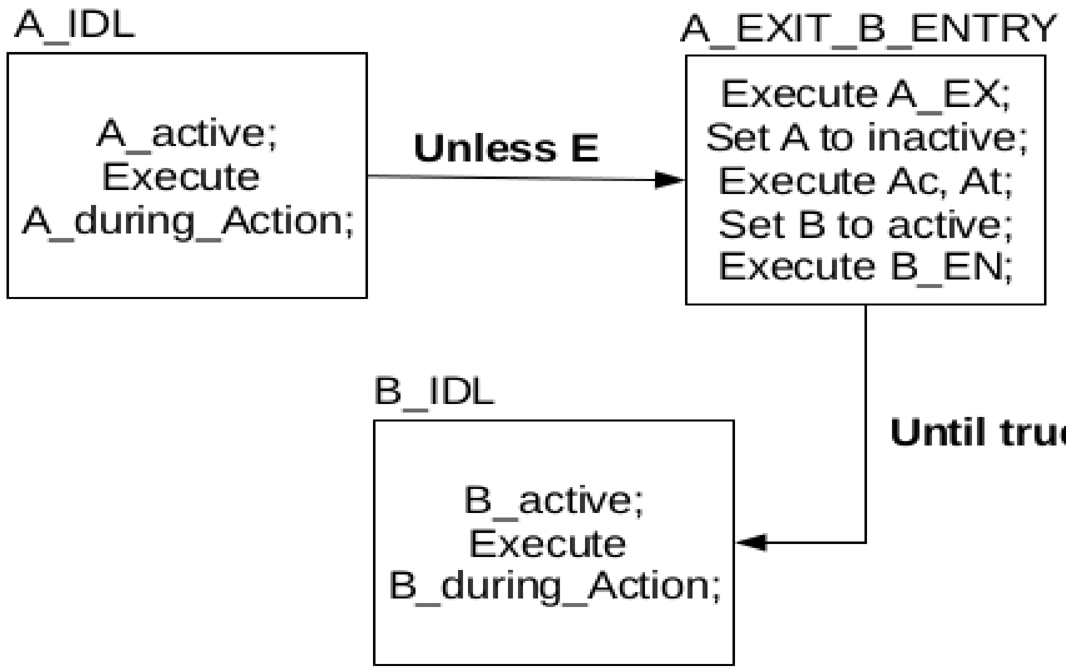
5.2 Translation into Automaton : First Version.

En : entry Action;
Ex : exit Action;
On-event : on event Action;

Stateflow example :

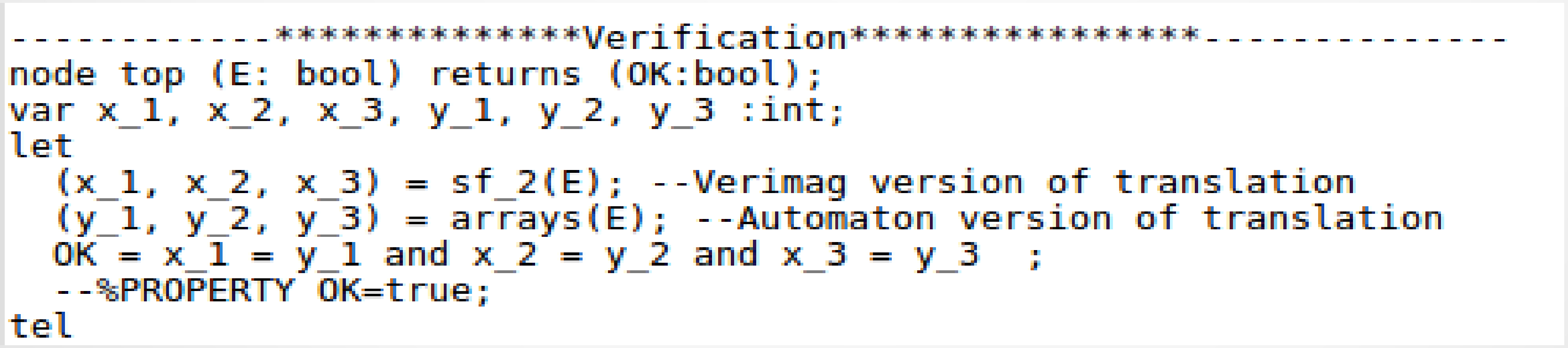


Automaton version :



5.2 Translation into Automaton : First Version.

The following examples are translated into our Automaton version and compared to Verimag Version. They have the same behavior and tested using our Lustrec compiler (top node).



```
------------**************Verification****************-------------
node top (E: bool) returns (OK:bool);
var x_1, x_2, x_3, y_1, y_2, y_3 :int;
let
  (x_1, x_2, x_3) = sf_2(E); --Verimag version of translation
  (y_1, y_2, y_3) = arrays(E); --Automaton version of translation
  OK = x_1 = y_1 and x_2 = y_2 and x_3 = y_3  ;
  --%PROPERTY OK=true;
tel
```

Switch Example

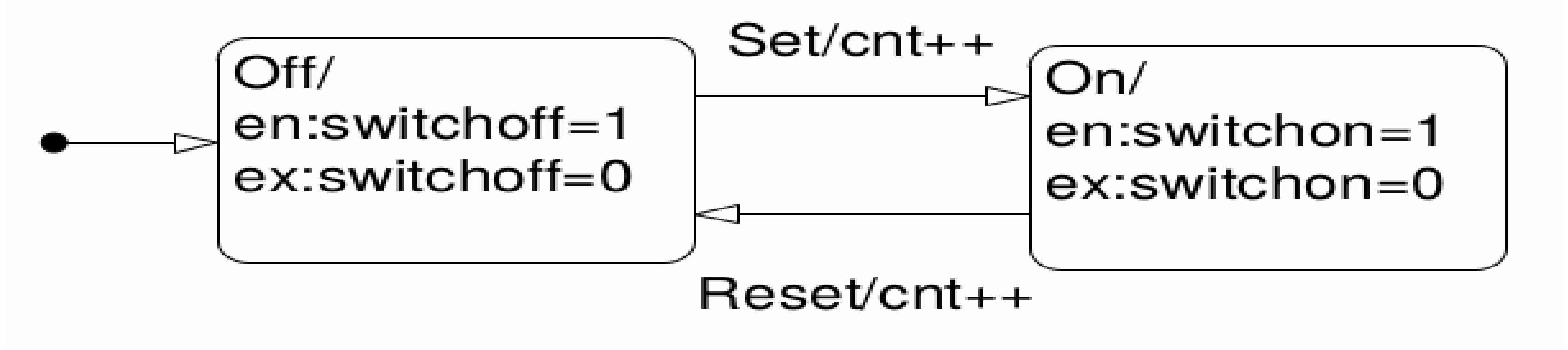
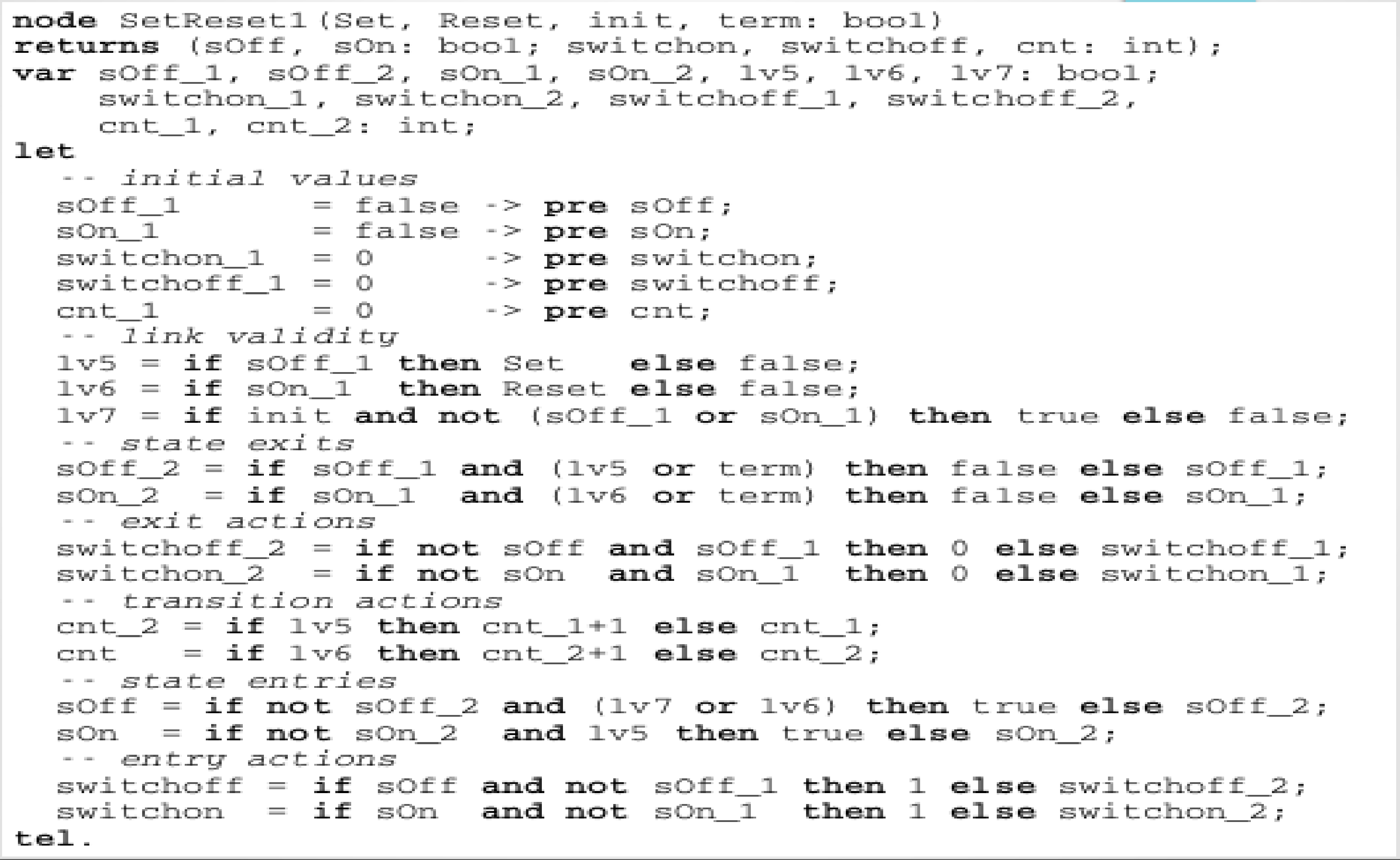


Figure 6: A simple STATEFLOW chart

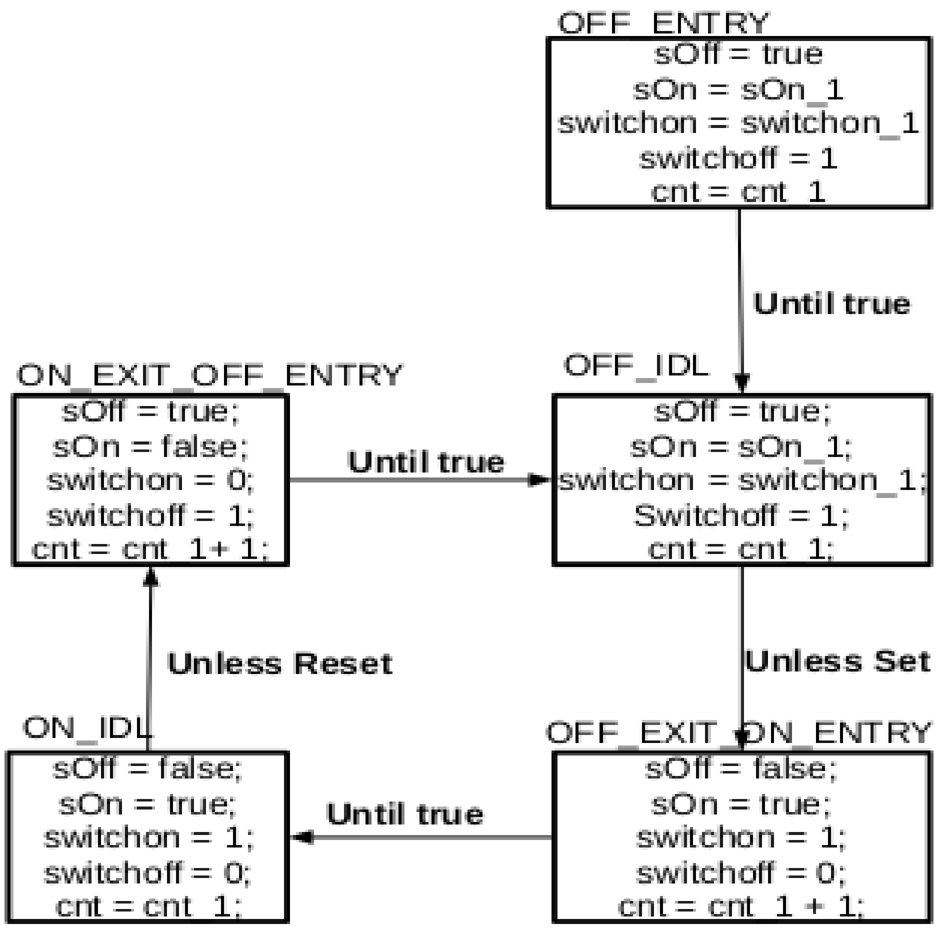
Switch example in Lustre (Verimag version)



```
node SetReset1(Set, Reset, init, term: bool)
returns (sOff, sOn: bool; switchon, switchoff, cnt: int);
var sOff_1, sOff_2, sOn_1, sOn_2, lv5, lv6, lv7: bool;
    switchon_1, switchon_2, switchoff_1, switchoff_2,
    cnt_1, cnt_2: int;
let
  -- initial values
  sOff_1      = false -> pre sOff;
  sOn_1       = false -> pre sOn;
  switchon_1  = 0     -> pre switchon;
  switchoff_1 = 0     -> pre switchoff;
  cnt_1       = 0     -> pre cnt;
  -- link validity
  lv5 = if sOff_1 then Set   else false;
  lv6 = if sOn_1  then Reset else false;
  lv7 = if init and not (sOff_1 or sOn_1) then true else false;
  -- state exits
  sOff_2 = if sOff_1 and (lv5 or term) then false else sOff_1;
  sOn_2  = if sOn_1  and (lv6 or term) then false else sOn_1;
  -- exit actions
  switchoff_2 = if not sOff and sOff_1 then 0 else switchoff_1;
  switchon_2  = if not sOn  and sOn_1  then 0 else switchon_1;
  -- transition actions
  cnt_2 = if lv5 then cnt_1+1 else cnt_1;
  cnt   = if lv6 then cnt_2+1 else cnt_2;
  -- state entries
  sOff  = if not sOff_2 and (lv7 or lv6) then true else sOff_2;
  sOn   = if not sOn_2  and lv5 then true else sOn_2;
  -- entry actions
  switchoff = if sOff and not sOff_1 then 1 else switchoff_2;
  switchon  = if sOn  and not sOn_1  then 1 else switchon_2;
tel.
```

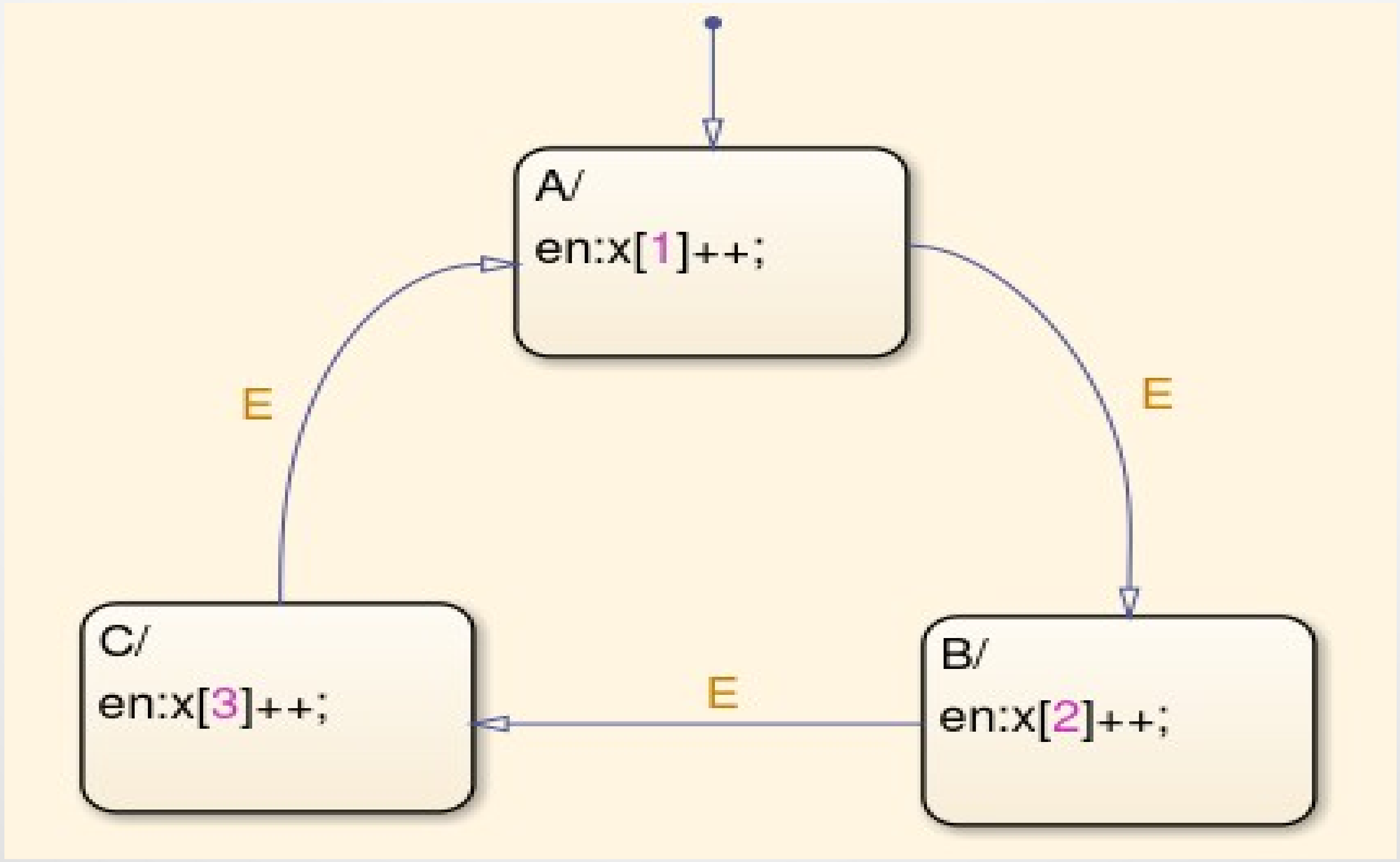
Switch example in Automaton version

```
-- initial values global values to all states
sOff_1 = false -> pre sOff;
sOn_1 = false -> pre sOn;
switchon_1 = 0 -> pre switchon;
switchoff_1 = 0 -> pre switchoff;
cnt_1 = 0 -> pre cnt;
```



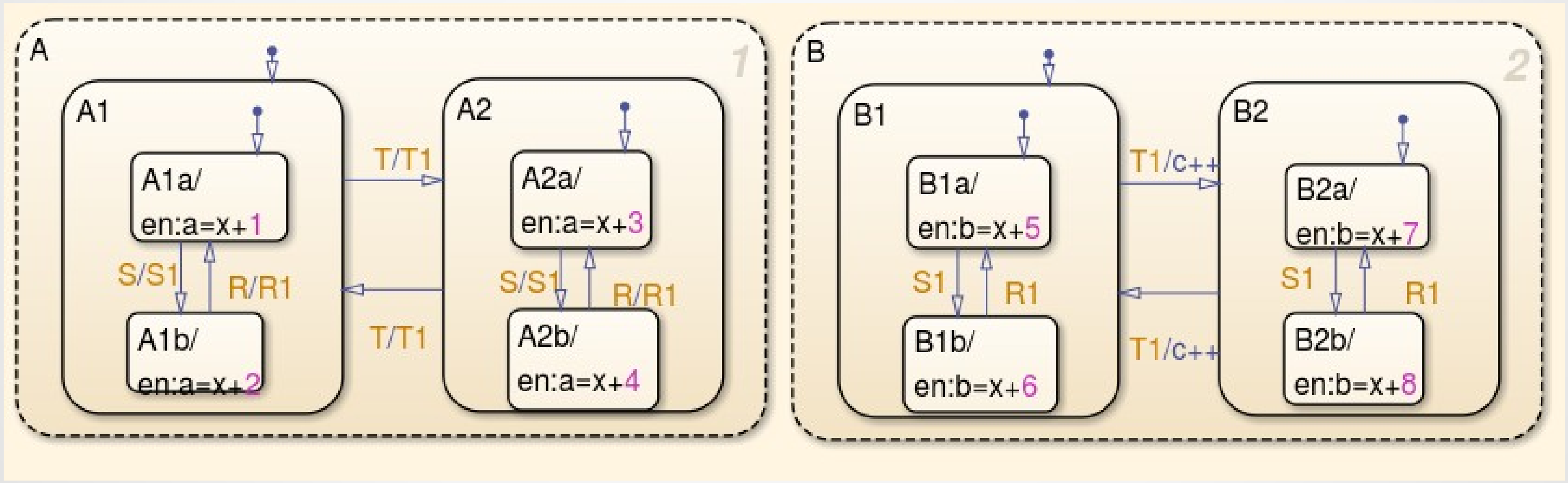
Arrays Example

In this example we use three integer outputs x1, x2 and x3 instead of an array x int^3.



Events Example

This example is not yet completely tested.
The output “a” is tested and valid.


A
1
A1
A1a/
en:a=x+1
S/S1
R/R1
A1b/
en:a=x+2
T/T1
T/T1
A2
A2a/
en:a=x+3
S/S1
R/R1
A2b/
en:a=x+4
B
2
B1
B1a/
en:b=x+5
S1
R1
B1b/
en:b=x+6
T1/c++
T1/c++
B2
B2a/
en:b=x+7
S1
R1
B2b/
en:b=x+8