MEC : a system for constructing and analysing transition systems

André Arnold
Laboratoire d'Informatique *
Université Bordeaux I

Abstract

MEC is a tool for constructing and analysing transition systems modelizing processes and systems of communicating processes.

From representations of processes by transition systems and from a representation of the interactions between the processes of a system by the set of all allowed global actions, MEC builds a transition system representing the global system of processes as the *synchronized product* of the component processes.

Such transition systems can be checked by computing sets of states and sets of transitions verifying properties given by the user of MEC. These properties are expressed in a language allowing definitions of new logical operators as least fixed points of systems of equations; thus all properties expressed in most of the branching-time temporal logic can be expressed in this language too.

MEC can handle transition systems with some hundred thousands states and transitions. Constructions of transition systems by synchronized products and computations of sets of states and transitions are performed in time linear with respect to the size of the transition system.

Introduction

The notion of *transition system* plays an important role for describing and studying processes and systems of communicating processes. A simple way to represent processes, introduced for instance in [10] and widely used in many works on semantics and verification of processes, is to consider that a process is a set of *states* and that an *action* or an *event* makes the current state of the process to change; thus the possible elementary behaviours of the process are represented by *transitions*: each transition contains the current state of the process, the new state it enters and the name of the action or event which caused this change. Transition systems are also used to describe systems of communicating processes, and not only individual processes; the states of the system are the tuples of states of its components and the transitions of the systems

*Unité de Recherche associée au Centre National de la Recherche Scientifique n° 726
†Work supported by the French Research Project C^3

are tuples of transitions of the components, provided these transitions are allowed – or obliged – to be executed simultaneously. Arnold and Nivat [12,3,1] have named this construction, which is implemented in MEC, *synchronization product*.

Once a system of processes is represented as a transition system, one can extract, from this transition system, some informations about the behaviour of the system of processes it represents. It is what we call *analysis* of a transition system and it amounts to computing the set of states or the set of transitions which satisfy some property of interest when looking at the behaviour of the system. For instance it is easy to check if the transition system has "deadlocks", i.e. states in which no transition is executable, or states in which every executable transition leads to a deadlock; a very simple algorithm can give the set of all these states. Thus an analyser is simply a tool which computes the set of all states or of all transitions of a given transition system satisfying some given property. The main feature of such an analyser is obviously the family of properties of states and transitions it can deal with.

In the systems **Cesar** [14] and **emc** [6], properties of states are expressed by formulas of branching time temporal logics. Given a formula F and a transition system $\mathcal{A}$, these systems compute the set $F_{\mathcal{A}}$ of states of $\mathcal{A}$ satisfying F (or, at least, decide if the "initial state" of $\mathcal{A}$ belongs to $F_{\mathcal{A}}$). In MEC we adopt a slightly different point of view, in some sense more algebraic than logical [7]. Let ω be some logical operator and let $F = \omega(F_1, \ldots, F_n)$ be a formula. For a transition system $\mathcal{A}$, the set $F_{\mathcal{A}}$ of states satisfying $\mathcal{A}$ depends on the sets $(F_i)_{\mathcal{A}}$ of states satisfying F_i, thus $F_{\mathcal{A}} = \omega_{\mathcal{A}}((F_1)_{\mathcal{A}}, \ldots, (F_n)_{\mathcal{A}})$, where $\omega_{\mathcal{A}}$ is an operator defined on the cartesian product of the powerset of states with the powerset of states as a range. Then formulas can be considered as expressions which have to be evaluated, in a way very similar to what happens in programming languages with arithmetic or boolean expressions. Thus the language used in MEC to express properties consists in variables and constants ranging over the powerset of states and on the powerset of transitions, and of sorted operators; the basic mechanism implemented in MEC is the execution of assignments *variable* := *expression*, exactly like in programming languages.

It remains to define the basic operators which can be used to build expressions. We can take the operators of branching time temporal logics (which are computable in linear time with respect to the size of the transition system) but, also, any other kind of operator which is easily computable. For instance in MEC we use the operator which associates with a set of states the union of the strongly connected components intersecting this set; although this operator is not really a logical operator, it is as easy to compute as the other ones, because of the Tarjan's algorithm [16] which is linear too.

Another feature of MEC is that the set of basic operators used to build expressions can be extended, in the same way that the set of operators in arithmetic expressions can be extended, in some programming languages, by defining new functions (especially recursive functions). It is well known that temporal logic operators can be characterized as least fixed points of equations [15,8,6], and this observation has led to the definition of the μ-calculus as an extension of branching time temporal logics [13,11]. MEC provides for the definition of new operators characterized as least fixed points of systems of equations [7] and then its expressive power is at least as powerful as the expressive power of alternation-depth-one μ-calculus defined by Emerson and Lei [9]. Indeed these new operators defined by systems of equations are still computable in linear time, like the basic ones, because of the Arnold-Crubillé's algorithm [2] to solve fixed point equations.

1 Transition systems

1.1 Labelled transition systems

A *labelled transition system* over an alphabet A of *actions* or *events* is a tuple $\mathcal{A} = < S, T, \alpha, \beta, \lambda >$ where

- S is a finite set of *states*,
- T is a finite set of *transitions*,
- $\alpha, \beta : T \longrightarrow S$ are the mappings which associate with every transition t its *source state* $\alpha(t)$ and its *target state* $\beta(t)$,
- $\lambda : T \longrightarrow A$ labels a transition t by the action or event $\lambda(t)$ which causes this transition.

We assume that there never exist two different transitions with the same label between the same two states, i.e. the mapping $< \alpha, \lambda, \beta >: T \longrightarrow S \times A \times S$ is injective.

1.2 Parametrized transition systems

A parametrized transition system is a labelled transition system given with some sets of designed states and some sets of designed transistions, called parameters. The role of these parameters is to give some additional informations on the transition system; it is the case when some states play a special role or when some transitions play a special role which is not specified by the label of the transition. Some example of such situations will be given below.

Example 1. Let us consider a boolean variable. It has two states, denoted by **0** and **1**, according to the current value (0 or 1) of the variable. The set A of actions performed by such a boolean variable contains

to0 which means that the variable is set to 0,

to1 which means that the variable is set to 1,

is0 which tests whether the value of the variable is 0,

is1 which tests whether the value of the variable is 1,

e which does nothing.

The first two actions modify the value of the variable, i.e. its state, in an obvious way. The two tests can be executed only if the variable has the tested value, and this value is not modified. The last action, when executed, does not change the value of the variable. As we shall see later on, (example 3), this null action is a way to express the possibility of occurrence of events which does not modify the state of the variable. The transition system representing this variable is given in figure 1, in the input format of MEC.

```
transition_system  b  < width = 0 >;

0 |- e      -> 0 ,
     to0    -> 0 ,
     to1    -> 1 ,
     is0    -> 0 ;

1 |- e      -> 1 ,
     to0    -> 0 ,
     to1    -> 1 ,
     is1    -> 1 ;

< initial = { 0 } >.
```

Figure 1: The transition system for a boolean variable in MEC

In this figure one can notice the last line `< initial = { 0 } >.` which defines a parameter, named "initial", reduced to a single state, **0**. This parameter is used to say that the state **0** has some special property, indeed it is the initial state of the transition system: the initial value of the variable, before any action is performed, is **0**.

Example 2. Let us now consider the Peterson's algorithm for mutual exclusion of two processes. This algorithm uses three shared boolean variables, `flag[0]`, `flag[1]`, and `turn`, all three initialised to **0**. Each one of the two processes executes the program given in figure 2 where `me` is equal to `0` and `other` is equal to `1` for the first process, and `me` is equal to `1` and `other` is equal to `0` for the second one. This program can also be represented by a transition system, states of which are the locations in the program and transitions between locations are labelled by elementary actions performed in the execution of the program. We also consider a special action **e** which does not change the state, which means that a process can stay idle at every moment. The MEC description of this transition system is given in figure 3.

In this transition system there are three state parameters:

initial which indicates the starting location,

cs which indicates the location where the process is in its critical section,

ncs which indicates the location where the process is not in its critical section.

There is also a transition parameter: it is the set of all transitions marked as having the property **mb**. These transitions are those executed by the processes when it tries to enter its critical section.

Indeed this transition system is obtained by interpreting the command `WAIT( ... OR ... )` in the following way: this command can be executed only if one of the two

```
proc( me , other ) =

while true do
begin
{NCS}          0: ... ;
{mutexbegin}      flag[me] := 1 ;
               1: turn = me ;
               2: WAIT (flag[other] = 0  OR turn = other) ;
{CS}           3: ... ;
{mutexend}        flag[me] := 0 ;
end
```

Figure 2: The Peterson algorithm

```
transition_system  proc  < width = 0 >;

0  |- e                -> 0 ,
      my_flag_to_1     -> 1 <property=(mb)>;

1  |- e                -> 1 <property=(mb)>,
      turn_to_me       -> 2 <property=(mb)>;

2  |- e                -> 2 <property=(mb)>,
      is_other_flag_0  -> 3 <property=(mb)>,
      is_turn_other    -> 3 <property=(mb)>;

3  |- e                -> 3 ,
      my_flag_to_0     -> 0 ;

< initial = { 0 } ; cs = { 3 } ; ncs = { 0 } >.
```

Figure 3: The transition system for a process in MEC

conditions is satisfied; in this case the execution of the process reaches the critical section. It looks like idle waiting. Another interpretation of this command (busy waiting) could be: the process tests the first condition; if it is true it reaches the critical section, otherwise it tests the second condition; if it is true it reaches the critical section, otherwise it executes WAIT again. This interpretation will yield another transition system.

2 Synchronized systems

Let us consider n transition systems $\mathcal{A}_i$ over the alphabets A_i of actions, for $i = 1, \ldots, n$. Let us assume that these transition systems represent processes and shared objects constituting a system of interacting processes.(For simplicity, from now on, we shall also call processes the shared objects since they are also represented by transition systems). That means that some action in some process can be executed only simultaneously with some other action in some other process, or, on the opposite, cannot be executed simultaneously with some other action of some other process. Let us call *global action* a vector $< a_1, \ldots, a_n >$ where a_i belongs to A_i. Such a global action is executed when the actions a_i are simultaneously executed by the n processes. Thus the interactions between the processes of a system can be represented by the set of all global actions which are allowed to be executed and in [3], it is advocated that this kind of specification of the interactions between the processes of the processes of a system is general enough to formalise most of the concurrent systems of processes.

2.1 Synchronization constraints

As explained above, the interactions between the processes $\mathcal{A}_i$ of a system are represented by a subset I of $A_1 \times \cdots \times A_n$, called a *synchronization constraint*.

Example 3. Let us consider again Peterson's mutual exclusion algorithm for two processes. It is a represented by a system containing two transition systems proc described in figure 3 and three boolean variables b described in figure 1. The second line of figure 4 gives the list of the transition systems of this system. The other lines are the elements of the synchronization constraint; these elements are just those we get when obeying the following rules. (Here we temporarily come back to the distinction between processes and variables).

1. We assume this system runs on a single processor; therefore the two processes cannot execute simultaneously a non null action; moreover the two processes cannot be idle simultaneously.

2. Each action performed by a process consists in setting or testing a variable. When a process executes such an action, the corresponding variable executes the corresponding action and the other variables execute the null action.

2.2 Synchronized product

Given a vector $< \mathcal{A}_1, \ldots, \mathcal{A}_n >$ of transition systems, each $\mathcal{A}_i = < S_i, T_i, \alpha_i, \beta_i, \lambda_i >$ over the alphabet A_i, the *free product* of $< \mathcal{A}_1, \ldots, \mathcal{A}_n >$ is the transition system

```
synchronization_system  peterson

< width = 5 ; list = (proc,proc,b,b,b) > ;

(my_flag_to_0     .e                .to0 .e    .e    ) ;
(my_flag_to_1     .e                .to1 .e    .e    ) ;
(e                .my_flag_to_0     .e   .to0  .e    ) ;
(e                .my_flag_to_1     .e   .to1  .e    ) ;
(turn_to_me       .e                .e   .e    .to0  ) ;
(e                .turn_to_me       .e   .e    .to1  ) ;
(is_other_flag_0  .e                .e   .is0  .e    ) ;
(e                .is_other_flag_0  .is0 .e    .e    ) ;
(is_turn_other    .e                .e   .e    .is1  ) ;
(e                .is_turn_other    .e   .e    .is0  ) .
```

Figure 4: The system representing Peterson's algorithm

$< S, T, \alpha, \beta, \lambda >$ over $A_1 \times \cdots \times A_n$ defined by

$$\begin{aligned} S &= S_1 \times \cdots \times S_n, \\ T &= T_1 \times \cdots \times T_n, \\ \alpha(t_1, \ldots, t_n) &= < \alpha_1(t_1), \ldots, \alpha_n(t_n) >, \\ \beta(t_1, \ldots, t_n) &= < \beta_1(t_1), \ldots, \beta_n(t_n) >, \\ \lambda(t_1, \ldots, t_n) &= < \lambda_1(t_1), \ldots, \lambda_n(t_n) > . \end{aligned}$$

In some sense, the free product represents the evolution of the vector of transition systems when no constraint is set on the actions which can be performed simultaneously. In case of a synchronization constraint some transitions of this free product will never appear : those which are labelled by a vector of actions not allowed by the synchronization constraint. Hence we have the following definition.

Given a vector $< \mathcal{A}_1, \ldots, \mathcal{A}_n >$ of transition systems, each $\mathcal{A}_i$ over the alphabet A_i, and a synchronization constraint I included in $A_1 \times \cdots \times A_n$, the *synchronized product* of $< \mathcal{A}_1, \ldots, \mathcal{A}_n >$ with respect to I is the transition system $< S, T_I, \alpha, \beta, \lambda >$ over $A_1 \times \cdots \times A_n$ where

- $< S, T, \alpha, \beta, \lambda >$ is the free product of $\mathcal{A}_1, \ldots, \mathcal{A}_n$;
- T_I is the set of transitions $t =< t_1, \ldots, t_n >$ of T having their label $\lambda(t) = < \lambda_1(t_1), \ldots, \lambda_n(t_n) >$ in I.

Indeed the synchronized product computed by MEC is only a sub-transition system of the synchronized system defined above. Each transition system $\mathcal{A}_i = < S_i, T_i, \alpha_i, \beta_i, \lambda_i >$ which is a component of a synchronization system is assumed to have a parameter `initial`, as it is the case for the transition systems described in examples 1 and 2. This parameter defines a subset $initial_i$ of S_i and the parameter `initial` of the product

is defined as the subset $initial = initial_1 \times \cdots \times initial_n$. The set of global states of the synchronized product of MEC is the set *Reach(initial)* of global states which can be reached from *initial*, i.e. the states of *initial* and the targets of paths having their sources in *initial*. The set of global transitions of the synchronized product of MEC is the set of global transitions having both their sources and their targets in *Reach(initial)*.

Example 4. The synchronized product, computed by MEC, of the synchronization system `peterson` given in figure 4 is given in figure 5. It was obtained by executing the MEC command `sync(peterson,res);` where `res` is the name given to the product.

3 Elementary computations

The general form of a computation command in MEC is

variable := expression;

the *expression* is evaluated and its value is assigned to the *variable*. The value of an expression is either a set of states or a set of transitions.

3.1 Set variables

Variables used by MEC are of two different sorts according to the kind of set they can be assigned. A variable is implicitely declared when it appears for the first time in the left hand part of an assignment command and its sort is the sort of the expression (if the sort of the expresssion can be unambiguously determined, otherwise the assignment is rejected). Every declared variable is displayed on the terminal with the number of objects (states or transitions) of its value. Parameters are considered as variables with an initial value.

3.2 Expressions

Expressions are built up from variables and operators. Among these operators are set-theoretical (or boolean) operators union, intersection, and difference as well as the constants "empty", denoted by {}, and "all", denoted by *.

Some others operators are primitive and will be described below. New operators can be defined by the users and this will be explained in the next section.

Finally there are some other ways to define sets of states and sets of transitions. We will not list all these ways here and we refer to the user manual [4].

Example 5. Let us consider the transition system `res` of figure 3 constructed by MEC, which will be our running example from now on.

First of all we want to know if the *mutual exclusion* property is verified, i.e. if the two processes can be or not both together in their critical section. Let us remind that the set of states in which a process is in its critical section is defined by the parameter `cs` as shown in figure 2. Therefore we have just to know whether there are (global) states in which

```
transition_system res          < width =          5> ;
e(0.0.0.0.0) |- (e.my_flag_to_1.e.to1.e) -> e(0.1.0.1.0) ,
                (my_flag_to_1.e.to1.e.e) -> e(1.0.1.0.0) ;
e(1.0.1.0.0) |- (e.my_flag_to_1.e.to1.e) -> e(1.1.1.1.0) ,
                (turn_to_me.e.e.e.to0) ->   e(2.0.1.0.0) ;
e(2.0.1.0.0) |- (e.my_flag_to_1.e.to1.e) -> e(2.1.1.1.0) ,
                (is_other_fla.e.e.is0.e) -> e(3.0.1.0.0) ;
e(3.0.1.0.0) |- (e.my_flag_to_1.e.to1.e) -> e(3.1.1.1.0) ,
                (my_flag_to_0.e.to0.e.e) -> e(0.0.0.0.0) ;
e(0.1.0.1.0) |- (e.turn_to_me.e.e.to1) ->   e(0.2.0.1.1) ,
                (my_flag_to_1.e.to1.e.e) -> e(1.1.1.1.0) ;
e(1.1.1.1.0) |- (e.turn_to_me.e.e.to1) ->   e(1.2.1.1.1) ,
                (turn_to_me.e.e.e.to0) ->   e(2.1.1.1.0) ;
e(2.1.1.1.0) |- (e.turn_to_me.e.e.to1) ->   e(2.2.1.1.1) ;
e(3.1.1.1.0) |- (e.turn_to_me.e.e.to1) ->   e(3.2.1.1.1) ,
                (my_flag_to_0.e.to0.e.e) -> e(0.1.0.1.0) ;
e(2.2.1.1.0) |- (e.is_turn_othe.e.e.is0) -> e(2.3.1.1.0) ;
e(2.3.1.1.0) |- (e.my_flag_to_0.e.to0.e) -> e(2.0.1.0.0) ;
e(0.0.0.0.1) |- (e.my_flag_to_1.e.to1.e) -> e(0.1.0.1.1) ,
                (my_flag_to_1.e.to1.e.e) -> e(1.0.1.0.1) ;
e(1.0.1.0.1) |- (e.my_flag_to_1.e.to1.e) -> e(1.1.1.1.1) ,
                (turn_to_me.e.e.e.to0) ->   e(2.0.1.0.0) ;
e(0.1.0.1.1) |- (e.turn_to_me.e.e.to1) ->   e(0.2.0.1.1) ,
                (my_flag_to_1.e.to1.e.e) -> e(1.1.1.1.1) ;
e(0.2.0.1.1) |- (e.is_other_fla.is0.e.e) -> e(0.3.0.1.1) ,
                (my_flag_to_1.e.to1.e.e) -> e(1.2.1.1.1) ;
e(0.3.0.1.1) |- (e.my_flag_to_0.e.to0.e) -> e(0.0.0.0.1) ,
                (my_flag_to_1.e.to1.e.e) -> e(1.3.1.1.1) ;
e(1.1.1.1.1) |- (e.turn_to_me.e.e.to1) ->   e(1.2.1.1.1) ,
                (turn_to_me.e.e.e.to0) ->   e(2.1.1.1.0) ;
e(1.2.1.1.1) |- (turn_to_me.e.e.e.to0) ->   e(2.2.1.1.0) ;
e(2.2.1.1.1) |- (is_turn_othe.e.e.e.is1) -> e(3.2.1.1.1) ;
e(3.2.1.1.1) |- (my_flag_to_0.e.to0.e.e) -> e(0.2.0.1.1) ;
e(1.3.1.1.1) |- (e.my_flag_to_0.e.to0.e) -> e(1.0.1.0.1) ,
                (turn_to_me.e.e.e.to0) ->   e(2.3.1.1.0);
< initial = { e(0.0.0.0.0) } >.
```

Figure 5: A synchronized product

(i) the first component is in the value of `cs` in the first transition system of the synchronization system `peterson`,

(ii) the second component is in the value of `cs` in the second transition system of the synchronization system `peterson`.

The sets of states satisfying (i) and (ii) are respectively denoted by `cs[1]` and `cs[2]`; hence the set of states not satisfying the mutual exclusion property is defined and/or computed by the assignment `nok := cs[1]/\cs[2];`

After execution of this command, it appears on the screen that the value of `nok` is a set of states which has 0 element. □

3.3 Primitive operators

Let us denote by σ and τ the sorts "set of states" and "set of transitions". We can use the following operators `src` of sort $\tau \rightarrow \sigma$, `tgt` of sort $\tau \rightarrow \sigma$, `rsrc` of sort $\sigma \rightarrow \tau$, `rtgt` of sort $\sigma \rightarrow \tau$. The interpretation of these operators is as follows, for a given transition system $< S, T, \alpha, \beta, \lambda >$: If Q is a set of states included in S and R a set of transitions included in T, then

$$\begin{aligned} \texttt{src}(R) &= \{\alpha(t) \mid t \in R\}, \\ \texttt{tgt}(R) &= \{\beta(t) \mid t \in R\}, \\ \texttt{rsrc}(Q) &= \{t \mid \alpha(t) \in Q\}, \\ \texttt{rtgt}(Q) &= \{t \mid \beta(t) \in Q\}. \end{aligned}$$

In other words $\texttt{src}(R)$ and $\texttt{tgt}(R)$ are respectively the sets of sources and targets of transitions in R and $\texttt{rsrc}(Q)$ and $\texttt{rtgt}(Q)$ are their reciprocals : the sets of transitions having their source and their target in Q.

Example 6. If Q is a set of states, the set *Pred(Q)* is the set of all states which are the source of a transition whose target is in Q. If `Q` is a variable whose value is Q, *Pred(Q)* is the value of the expression `src(rtgt(Q))`.

In a similar way *Succ(Q)* is the value of the expression `tgt(rsrc(Q))`.

If `T` denotes a set T of transitions, the expression `src(T/\rtgt(Q))` evaluates to the set of sources of transitions in T having their target in Q and `tgt(T/\rsrc(Q))` to the set of targets of transitions in T having their source in Q. □

We also use the binary operator `loop` of sort $\tau\tau \rightarrow \tau$ defined by $t \in \texttt{loop}(R, R')$ if and only if t belongs to a path p such that

(i) the source of p is equal to its target,

(ii) every transition of p is in R',

(iii) some transition of p is in R,

In other words, a transition t is in $\texttt{loop}(R, R')$ if it belongs to some loop in R' containing some transition in R.

Example 7. Let us now look for a *livelock* in the transition system **res**.

Roughly speaking there is a livelock if there is an infinite execution where

(i) both processes are always in their "mutexbegin", and

(ii) none of the processes remains "inactive" forever.

Condition (i) means that both processes are trying to enter their critical section and never succeed; condition (ii) means that both processes are really trying to enter their critical section : if one of the processes stays idle forever surely it will not enter its critical section and could even prevent the other process to enter.

Since the only waiting action is denoted by `e`, a process is active during a transition t if the corresponding component of the label of this transition is not equal to `e`. Therefore the sets of transitions in which the first and the second processes are active are computed by `active1 := !label[1] # "e";` and `active2 := !label[2] # "e";` The set of transitions in which both processes are in their "mutexbegin" is computed by `ll := mb[1]/\mb[2];`

If there is a livelock, there is a "loop" p such that

(i) all its transitions are in `ll`;

(ii) it has an infinite number of transitions in `active1`;

(iii) it has an infinite number of transitions in `active2`.

The set `ll0` of transitions belonging to a loop satisfying (i) is computed by `ll0 := loop(*,ll);` the set `ll1` of transitions belonging to a loop satisfying (i) and (ii) is computed by `ll1 := loop(active1,ll0);` finally the set `ll2` of transitions belonging to a loop satisfying (i),(ii) and (iii) is computed by `ll2 := loop(active2,ll1);` Here again this set is empty. □

4 The definition of new operators

4.1 Preliminary example

Let us consider some given transition system $\mathcal{A}$. Let us consider the set *Reach(initial)* which was used in the definition of the synchronized product (cf. 2.2). Indeed for every set Q of states one can define the set *Reach(Q)* containing Q and the targets of paths having their source in Q. Thus *Reach* can be considered as an operator of sort $\sigma \to \sigma$ and one can think of adding it to the primitive operators.

But we can also remark that this operator can be formally defined in the following way. Let us consider the operator *Succ* defined in example 6. We have the following equality:

$$Reach(Q) = Q \cup Succ(Reach(Q)) \tag{1}$$

We have even more: not only *Reach(Q)* is a solution of the equation

$$X = Q \cup Succ(X) \tag{2}$$

but it is the least set of states (for inclusion) satisfying this equation.

Therefore we can give *Reach* the following definition: for every set Q of states, *Reach(Q)* is the least solution of (2). This definition is expressed in MEC by

```
function reach(Q:state) return X:state;
begin
X = Q \/ tgt(rsrc(X))
end.
```

Once this function is defined, the operator `reach` can be used in expressions exactly like primitive operators; its sort is $\sigma \rightarrow \sigma$, as expressed by the first line of the definition.

4.2 Systems of equations

We are now going to formally define the systems of equations which can be used to define new operators in MEC.

Basic operators Let D be the heterogenous algebraic signature with two sorts, σ and τ, containing the following operators [7] :

0_σ, 1_σ, 0_τ, 1_τ : constants of sorts σ and τ;

$\cup_\sigma$, $\cap_\sigma$, $-_\sigma$: binary operators of sort $\sigma\sigma \longrightarrow \sigma$;

$\cup_\tau$, $\cap_\tau$, $-_\tau$: binary operators of sort $\tau\tau \longrightarrow \tau$;

src, *tgt* : binary operators of sort $\tau \longrightarrow \sigma$;

rsrc, *rtgt* : binary operators of sort $\sigma \longrightarrow \tau$.

If $\mathcal{A} =< S, T, \alpha, \beta, \lambda >$ is a transition system, it is given a D-structure in the following way :

- The set of elements of sort σ (resp. τ) is $\wp(S)$ (resp. $\wp(T)$).
- The interpretations of the operators in a transition system $\mathcal{A}$ have been previously defined.

  All these operators, but $-_\sigma$ and $-_\tau$, have monotonic interpretations with respect to set inclusion.

Signed terms Let $X_n = \{x_1, \ldots, x_n\}$ and $Y_m = \{y_1, \ldots, y_m\}$ be two sets of variables of sort σ and τ. We can build terms with these variables and the operators of D. If t is such a term, its interpretation $t_\mathcal{A}$ will be a mapping from $\wp(S)^n \times \wp(T)^m$ into $\wp(S)$ or $\wp(T)$ according to the sort of this term.

Since the interpretation of a difference operator is not monotonic, the interpretation of a term is not necessarily monotonic. However we can consider that the interpretation of a difference becomes monotonic if its first argument is ordered by inclusion and its second argument is ordered by the inverse relation : containment.

This led us to consider two kinds of order on $\wp(S)$ and $\wp(T)$, inclusion and containment. We shall denote these powersets by $\wp^+(S)$ and $\wp^+(T)$ (resp. $\wp^-(S)$ and $\wp^-(T)$)

when we wish to make clear that they are ordered by inclusion (resp. containment). For each sort, we consider two kinds of variables : positive variables ranging over a powerset ordered by inclusion and negative variables ranging over a powerset ordered by containment. Let X^+ and X^- be two sets of positive and negative variables of sort σ, Y^+ and Y^- two sets of positive and negative variables of sort τ and Z_σ et Z_τ two set of "parameters", which will be interpreted as arbitrary sets.

We inductively define the sets of positive and negative terms of sort σ, $\mathcal{T}_\sigma^+$ et $\mathcal{T}_\sigma^-$, and of positive and negative terms of τ, $\mathcal{T}_\tau^+$ and $\mathcal{T}_\tau^-$ by

- $X^+ \subset \mathcal{T}_\sigma^+$, $X^- \subset \mathcal{T}_\sigma^-$, $Y^+ \subset \mathcal{T}_\tau^+$, $Y^- \subset \mathcal{T}_\tau^-$;
- $\{0_\rho, 1_\rho\} \cup Z_\rho \subset \mathcal{T}_\rho^+ \cap \mathcal{T}_\rho^-$; $(\rho = \sigma, \tau)$;
- if t_1 and t_2 belong to $\mathcal{T}_\rho^\varsigma$ then $t_1 \cup_\rho t_2$, $t_1 \cap_\rho t_2$ belong to $\mathcal{T}_\rho^\varsigma$; $(\rho = \sigma, \tau;\ \varsigma = +, -)$;
- if t belongs to $\mathcal{T}_\tau^\varsigma$ then $src(t)$ and $tgt(t)$ belong to $\mathcal{T}_\sigma^\varsigma$; $(\varsigma = +, -)$;
- if t belongs to $\mathcal{T}_\sigma^\varsigma$ then $rsrc(t)$ and $rtgt(t)$ belong to $\mathcal{T}_\tau^\varsigma$; $(\varsigma = +, -)$;
- if t_1 belongs to $\mathcal{T}_\rho^\varsigma$ and t_2 belongs to $\mathcal{T}_\rho^{\varsigma'}$ then $t_1 -_\rho t_2$ belongs to $\mathcal{T}_\rho^\varsigma$; $(\rho = \sigma, \tau;$ $< \varsigma, \varsigma' >=< +, - >,\ < -, + >)$;

If t is a term of $\mathcal{T}_\rho^\varsigma$ its interpretation $t_\mathcal{A}$ is then monotonic or antimonotonic (according to the value of ς) when the values of variables in Z are fixed and values of other variables are ordered according to their sign.

Example 8. If Z_τ is a parameter of sort τ, Z_σ a parameter of sort σ, X_+ a positive variable of sort σ and Y_- a negative variable of sort τ, then $Z_\tau \cap_\tau rtgt(1_\sigma -_\sigma X_+)$ is a negative term of sort τ and $Z_\sigma \cup_\sigma (1_\tau -_\tau src(Y_-))$ is a positive term of sort σ. □

Systems of equations Let us consider the following sets of variables :

$$\begin{aligned}
\mathbf{X}^+ &= \{X_1^+, \ldots, X_n^+\},\\
\mathbf{X}^- &= \{X_1^-, \ldots, X_{n'}^-\},\\
\mathbf{Y}^+ &= \{Y_1^+, \ldots, Y_m^+\},\\
\mathbf{Y}^- &= \{Y_1^-, \ldots, Y_{m'}^-\},\\
\mathbf{Z}_\sigma &= \{Z_1, \ldots, Z_p\},\\
\mathbf{Z}_\tau &= \{Z'_1, \ldots, Z'_{p'}\},
\end{aligned}$$

We consider the sets of terms $\mathcal{T}_\sigma^+$, $\mathcal{T}_\sigma^-$,$\mathcal{T}_\tau^+$,$\mathcal{T}_\tau^-$ built with these variables.

A system of equations Σ is

$$\begin{aligned}
&\{X_i^+ = u_i^+ | 1 \le i \le n\} \quad \cup \quad \{X_i^- = u_i^- | 1 \le i \le n'\} \quad \cup\\
&\{Y_i^+ = v_i^+ | 1 \le i \le m\} \quad \cup \quad \{Y_i^- = v_i^- | 1 \le i \le m'\}.
\end{aligned}$$

where $u_i^+ \in \mathcal{T}_\sigma^+$, $u_i^- \in \mathcal{T}_\sigma^-$, $v_i^+ \in \mathcal{T}_\tau^+$, $v_i^- \in \mathcal{T}_\tau^-$.

With a system of equations Σ and a transition system $\mathcal{A}$ we associate the ordered set

$$D_1 \;=\; \wp^+(S)^n \times \wp^-(S)^{n'} \times \wp^+(T)^m \times \wp^-(T)^{m'}$$

and the set

$$D_2 \;=\; \wp(S)^p \times \wp(T)^{p'}$$

ordered by the empty order. Then Σ defines a mapping

$$\Sigma_{\mathcal{A}} : D_1 \times D_2 \longrightarrow D_1$$

This mapping is monotonic with respect to the order defined componentwise on D_1 and $D_1 \times D_2$. Thus it has a least fixed point $\mu\Sigma_{\mathcal{A}} : D_2 \longrightarrow D_1$.

Now if we choose one of the signed variables, by composing $\mu\Sigma_{\mathcal{A}}$ with the projection of D_1 on its component associated with this variable, we get a mapping from D_2 in $\wp(S)$ or $\wp(T)$, according to the sort of the variable. Thus we can assume that a new operator is defined by :

- the list of sorted parameters,
- the list of sorted and signed variables,
- the selected variable defining the result,
- the list of equations, one for each variables.

The interpretation of such an operator in any transition system will be the mapping defined above.

Example 9. Let us consider the two terms of the example 8. The parameters they contains are Z_τ and Z_σ, and the variables are X_+ and Y_-. Let us consider the two equations

$$\begin{aligned} X_+ &= Z_\sigma \cup_\sigma (1_\tau -_\tau src(Y_-)) \\ Y_- &= Z_\tau \cap_\tau rtgt(1_\sigma -_\sigma X_+) \end{aligned}$$

For every transition system $\mathcal{A}$ this defines a mapping from $\wp(T) \times \wp(S)$ in $\wp(S) \times \wp(T)$; if we choose X_+ as principal variable we get a mapping from $\wp(T) \times \wp(S)$ in $\wp(S)$.

Let us call `unavoidable` this operator for some reasons which will be explained below. In MEC its definition will be written

```
function unavoidable(Zt:trans ; Zs:state) return X:state;
var Y:_trans
begin
X = Zs \/ (* - src(Y));
Y = Zt /\ rtgt(* - X)
end.
```

Let us remark that negative variables are specified by an "underscore" preceding their sort.

The name "unavoidable" given to this operator comes from the following property. Let $\mathcal{A}$ be any transition system, Q some set of states and R some set of transitions. Then a state s belongs to $\texttt{unavoidable}(R, Q)$ if and only if every maximal path in $\mathcal{A}$ (i.e. an infinite path or a finite path whose last state has no successor in $\mathcal{A}$) originated in s and containing only transitions in R contains a state in Q. In particular $\texttt{unavoidable}(T, \emptyset)$ is the set of states s such that every maximal path originated in s is finite. This can be considered as a definition of "deadlocking" states, since once in such a state it is impossible to start an infinite computation.

Computation of least fixed points If an expression contains an operator defined by a system of equation it remains to evaluate this expression. This amounts to computing the value of $\texttt{op}(Z_1, \ldots Z_n)$ when the values of $Z_1, \ldots, Z_n$ are known and thus to computing the least fixed point of the equations defining op when the parameters occuring in these equations are given. This can be done in time linear with respect to the size of the transition system (i.e. number of states and number of transitions, using an algorithm described in [2]). Thus all computations performed by MEC are done in a time linear with the size of the transition system.

5 An example of use

MEC has been used to check some mutual exclusion algorithms. It allowed to discover that Burns's algorithms [5] contained livelocks in the case of four processes.

The transition system obtained by synchronizing four processes, four boolean flags and a "turn" variable, representing the symmetrical Burns's algorithm with four processes has 65 016 states, 260 064 transitions stored in about 13 Mbytes of memory. On a Sun 3/60, it takes 20 minutes of CPU to construct this product and 11 minutes to compute `unavoidable` using the linear algorithm of Arnold and Crubillé.

References

[1] A. Arnold. Transition systems and concurrent processes. In *Mathematical problems in Computation theory (Banach Center Publications,vol. 21)*, 1987.

[2] A. Arnold and P. Crubillé. A linear algorithm to solve fixed point equations on transition systems. *Inf. Process. Lett.*, 29:57–66, 1988.

[3] A. Arnold and M. Nivat. Comportements de processus. In *Colloque AFCET "Les Mathématiques de l'Informatique"*, pages 35–68, 1982.

[4] D. Bégay. *Mode d'emploi MEC.* Technical Report I-8915, Université Bordeaux I, 1989.

[5] J. E. Burns. Symmetry in systems of asynchronous processes. In *Proc. 22nd Annual Symp. on Foundations of Computer Science*, pages 169–174, 1981.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logics specifications. *ACM Trans. Prog. Lang. Syst.*, 8:244–263, 1986.

[7] A. Dicky. An algebraic and algorithmic method for analysing transition systems. *Theoretical Comput. Sci.*, 46:285–303, 1986.

[8] E. A. Emerson and E. C. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. de Bakker and J. van Leeuwen, editors, *7th Int. Coll. on Automata, Languages and Programming*, pages 169–181, Lect. Notes. Comput. Sci. 85, 1980.

[9] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Symp. on Logic in Comput. Sci.*, pages 267–278, 1986.

[10] R. M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19:371–384, 1976.

[11] D. Kozen. Results on the propositional μ-calculus. *Theoretical Comput. Sci.*, 27:333–354, 1983.

[12] M. Nivat. Sur la synchronisation des processus. *Revue Technique Thomson-CSF*, 11:899–919, 1979.

[13] V. Pratt. A decidable μ-calculus. In *Proc. 22nd Symp. on Foundations of Comput. Sci.*, pages 421–427, 1981.

[14] J.-P. Queille. *Le système CESAR: Description, spécification et analyse des applications réparties.* PhD thesis, I.N.P., Grenoble, 1982.

[15] J. Sifakis. *Global and local invariants in transition systems.* Technical Report 274, IMAG, Grenoble, 1981.

[16] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.