The superimposition of ESTELLE programs : a tool for the implementation of observation and control algorithms

Benoit Caillaud

INRIA

UNITE DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1102

Programme 3
Réseaux et Systèmes Répartis

THE SUPERIMPOSITION OF ESTELLE PROGRAMS : A TOOL FOR THE IMPLEMENTATION OF OBSERVATION AND CONTROL ALGORITHMS

Benoît CAILLAUD

Octobre 1989

RR_1102

The superimposition of Estelle programs : a tool for the implementation of observation of observation and control algorithms

Benoît CAILLAUD

Publication Interne n° 493

Septembre 1989

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F

The superimposition of Estelle programs: A tool for the implementation of observation and control algorithms [1]

Publication Interne n° 493 - 30 Pages

Benoît Caillaud
E-mail : caillaud@irisa.fr

Abstract

The superimposition is a distributed program composition. It is a convenient concept for the design and implementation of control and observation algorithms in distributed systems, such as snapshots, detection of termination, global time, verification of properties, mutual exclusion, garbage collection. The present report describes the implementation of superimposition on static Estelle. It consists of a compiler that transforms a program in static Estelle extended to the superimposition into a pure static Estelle program. The problem of the correctness and complexity of the generated code is also raised.

La superposition de programmes Estelle : Un outil pour l'implantation d'algorithmes d'observation et de contrôle

Résumé

La superposition est une composition de programmes distribués bien adaptée à la conception et à l'implantation d'algorithmes de contrôle et d'observation de systèmes distribués (états globaux, détection de la terminaison, temps global, vérification de propriétés, exclusion mutelle, ramasse miettes). Ce rapport décrit l'implantation de la superposition dans le langage Estelle statique. Elle consiste en un compilateur qui transforme un programme en Estelle statique étendu à la superposition, en un programme en Estelle statique pur. Le problème de la correction et de la complexité du code engendré est aussi abordé.

[1] This work has been done in the team "Algorithmes Distribués et Protocoles" of the IRISA and is partially supported by the PRC-GRECO C^3.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Contents

1 Introduction

In several paradigms of the distributed algorithmics, the same concept arises: a program observes or controls another underlying program. This composition has been first studied in [3].

A compositional way of programming is of the greatest interest since one can deduce the properties of the global system from the proofs of each component of the composed program (let us call it a complex) and from a global invariant [5, 4].

As it will be explained more precisely in section 2, the superimposition of two programs over the same network behaves as if the two programs were confined in two independent closed layers, except that: firstly, each process of the upper layer has a read only, asynchronous access to the variables of the associated underlying process; secondly, some pairs of similar events of respectively an upper process and its associated lower process are synchronized (i.e. the underlying process can send a message if and only if the upper process sends a given message over the same edge of the network at the same time).

It is easy to show that the *partial correctness* of the underlying program remains the same when it is placed in a superimposed complex [4, 3]. Therefore this composition can be applied to the observation algorithms, whose properties are at least not to change the partial correctness of the observed program. The *total correctness* is much harder but not impossible to prove [4]. It implies the use of *fairness* [6], and the introduction of the concept of *freezing* [3]. More precisely one wants the upper program not to freeze the underlying one for an unbounded time.

Besides, this composition also gives the possibility to forbid some events, for instance communications, therefore controlling the underlying algorithm.

After a short recall on superimposition we will define static Estelle and its extension to superimposition. Then with the help of a few examples we will show that programming with superimposition is definitely easy. Before we explain the compiler, the transformation method will be detailed.

2 What is superimposition ?

The goal is not there to define superimposition completely, in an axiomatic way, but rather to give the reader some highlights on that concept. One can find a complete description of superimposition in [3, 4].

Let P and Q be two distributed programs over the same network $G = (V, E)$. Therefore, for any $i \in V$ we have a pair of processes (P_i, Q_i) that are respectively part of P and Q.

Let K be a one to one mapping[2] from a subset Σ^u of the set Λ^u of well-defined events[3] of P (the domain of K) into a subset $\Sigma^d \subset \Lambda^d$ of well-defined events of Q (the range of K).

The superimposition of P over Q relatively to K is the distributed program $S \equiv \frac{P}{Q}_K$

[2] K is also represented as a subset of $\Lambda^u \times \Lambda^d$, so that it defines the one to one mapping $\Sigma^u \subset \Lambda^u \longrightarrow \Sigma^d \subset \Lambda^d$

[3] These events can be composed events such as: P_i sends m over α or P_i sends m' over α. But the components of such an event must all be similar (all of them are emissions or receptions, over the same edge).

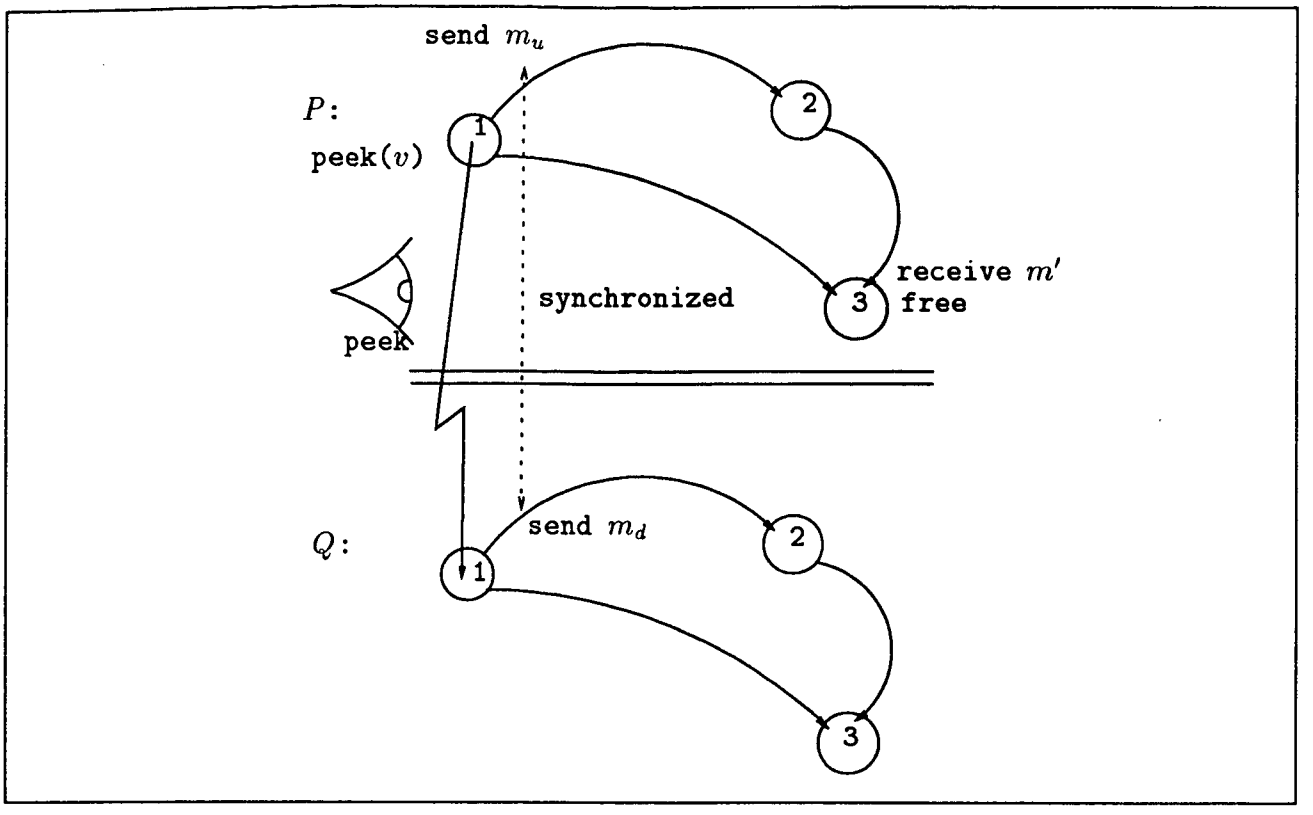

Figure 1: (send m_u, send m_d) $\in K$, the two message *send* are synchronized. receive m' is not synchronized with any event of Q_2 since $m' \notin \Sigma^u$.

which is the parallel composition of the processes $S_i \equiv \frac{P_i}{Q_i}_K$.

$$S \equiv \frac{\|_{i \in V} P_i}{\|_{i \in V} Q_i}_K \equiv \|_{i \in V} \frac{P_i}{Q_i}_K$$

The superimposition of two processes P_i and Q_i, both placed on the same vertex i of the network G is defined as follows:

- The processes P_i and Q_i are running in *parallel*.
- The two programs P and Q are *confined* in two distinct layers so that any P_i (resp. Q_i) cannot communicate (*send* or *receive* messages) with any process not belonging to its layer; then it can communicate only to a P_j (resp. Q_j) ($(i,j) \in E$ if it is a *send* and $(j,i) \in E$ if it is a *receive*).
- P_i and Q_i share the same memory, so that P_i can read *asynchronously* some variables of Q_i (this mecanism is called "*peek*").

- Any event ϵ (for instance a communication) in the domain of K (resp. range of K) is *synchronized* with $K(\epsilon)$ (resp. $K^{-1}(\epsilon)$).

These four rules are summarized in figure 1. The graph represents the network, where some events are occuring.

3 Static Estelle and its extension to superimposition

This composition in its distributed[4] form $\|_{i \in V} \frac{P_i}{Q_i}_K$ has been added to a subset of Estelle: Static Estelle, which is Estelle with the following constraint: *The parent of any process is inactive.*

In static Estelle, the network (the processes and the channels) is defined during the initialization part of the program, and cannot be changed afterwards, since the parents of the processes are inactive. Therefore there are two kinds of body:

- The refinement body which has only an *initialize* part and can hold some processes but cannot have any transition nor state declaration statement.
- The terminal body which is a pure process without any *body*, *module*, or *modvar* declared inside.

Integrating the distributed superimposition into Estelle simply means that firstly one adds a constructor for the superimposition of two modules P and Q, relatively to K, therefore defining a new module $S \equiv \frac{P}{Q}_K$. And secondly we have to describe the bodies that match this new module. Such a body is the superimposition of two terminal bodies matching respectively the modules P and Q.

3.1 Superimposing modules

The mapping K is defined in the following way:

- We need to declare the pairs of bounded interaction points that hold synchronized events. They must be compatible with each other (i.e. if one of them is an array of interaction points, then the other one must be also an array, with the *same* index type). Each pair of bounded interaction points defines a new interaction point of the module S.
- Some pairs of messages respectively from the channel of the upper interaction point, and from the channel of the lower interaction point are synchronized. The following consistency rule insures that K is well-defined: the roles of two synchronized messages must be either the respective roles of the interaction points or their conjugates.

[4] *Distributed* is here in its algebraic meaning: $a(b+c) = ab + ac$, where the right hand part of the equality is the distributed form.

So, we can give a syntax for the extension of Estelle:

```
module module-name ( formal-parameter-list );
        superimpose  module-up ( p-list-up )
        over         module-down ( p-list-down );
                     | bind  ip-up over ip-down |
        { ip { port : | up    ip-up             | ; }*}
                     | down  ip-down           |
        { compose ip-up.message-up over ip-down.message-down ; }*
end;
```

Where *p-list-up* and *p-list-down* are two sublists of the formal parameter list of the superimposed module. These allow us to share the parameters passed at the *init* time between the two constitutives modules, *module-up* and *module-down*.

The free interaction points of the upper (resp. lower) module *module-up* (resp. *module-down*) are renamed with the **up** (resp. **down**) construction. All ports of *module-up* and *module-down* must appear exactly once.

Then we have to define the bodies that will match the superimposed module.

3.2 Superimposing bodies

Such a body is the superimposition of two bodies, respectively for the upper and lower modules of the complex. And then, the last thing to be declared is the set of variables that are subject to *peeks*.

The syntax is:

```
body  body for  module ;
      superimpose  body-up over  body-down ;
      { peek  { var-up :=  var-down ;  }*}
end;
```

var-up must be a variable of the upper body, declared in the **var** field of the body. And *var-down* must also be a variable of the lower body, with the same scope. They must be assignment type compatible as the syntax prompts it.

Then any reference to *var-up* is actually a reference to *var-down*.

In order to follow the specifications of superimposition it is forbidden to put *var-up* neither on the left hand side of an assignement nor as a variable parameter of a procedure or function call.

There is no difference between a simple and a composed module or body. The use of *init, connect, attach,* etc is exactly the same.

4 A few examples of superimposed programs

The aim of this section is to give a few examples on the usage of superimposition. That's why the properties of the described algorithms are not proved. Anyway the proofs are in the corresponding citations. The programs are not completely detailed. Only their skeletons are given.

4.1 The Fidge and Mattern's clock

The first example is the global-clock of J. Fidge and F. Mattern [10] which can be directly implemented using the superimposition.

4.1.1 The algorithm

The clock is a vector of N^n, with the canonical partial order:

$$u \leq v \iff \forall i \in \{1, \ldots, n\}\ u[i] \leq v[i]$$

Let us denote $u \sqcup v$ the vector whose i-th component is $\max(u[i], v[i])$, and e_i the vector defined by: $\forall j \neq i\ e_i[j] = 0$ and $e_i[i] = 1$.

The time stamp $\Theta(m)$ associated to every message *send* or *receive* m of a n-processes distributed system is computed as follows:

- Initially on every process i, the local clock v_i is set to 0.
- For each event on the process i, we perform: $v_i := v_i + e_i$. The new v_i is the time stamp of the event.
- Every message is associated with the time-stamp of the sending process at the *send* time.
- At the receipt of a message with the time stamp u on process i: $v_i := u \sqcup v_i$, the new v_i is the time stamp of the message receipt.

It can be shown that this clock has good properties:

Theorem 1 *For every pair of events* ϵ, ν

$$\epsilon \textit{ is before } \nu \iff \Theta(\epsilon) \leq \Theta(\nu)$$

4.1.2 The program

Let us assume that the module descriptor of the calculus process is:

```
module calculus ( me : site_id );
  ip c_in  : array[site_id] of calculus_channel(calculus_in);
     c_out : array[site_id] of calculus_channel(calculus_out);
end;
```

and that the only message type of `calculus_channel` is `msg`. Then the observer module is:

```
module clock ( me : site_id );
  ip o_in  : array[site_id] of observer_channel(obs_in);
     o_out : array[site_id] of observer_channel(obs_out);
end;
```

assuming that we have previously declared:

```
channel observer_channel(obs_in,obs_out);
  by obs_out : stamp ( time : vector );
```

The superimposed module is:

```
module complex ( me : site_id );
  superimpose clock(me) over calculus(me);
  ip k_in  : bind o_in  over c_in;
     k_out : bind o_out over c_out;
  compose o_in.stamp  over c_in.msg;
  compose o_out.stamp over c_out.msg;
end;
```

and the body for `clock` is:

```
body b_clock for clock;
  var my_stamp : vector;

  initialize
    begin
      my_stamp := 0 (* vector *)
    end;

  trans any k : site_id do
        when o_in[k].stamp(time)
        begin
          my_stamp:=sup(my_stamp+e(me),time)
        end;

  trans any k : site_id do
        begin
          output o_out[k].stamp(my_stamp+e(me));
          my_stamp:=my_stamp+e(me)
        end;
end;
```

Note that the output statement is the first statement of the body of this transition, this is because it has a slightly different semantics: this transition can be fired only when the output is possible —i.e. the underlying program is ready to make a similar output. This change was necessary since we need *external choice*[5] for observers. As it will be explained in detail in section 5, we chose not to change the syntax of the transition, therefore giving a particular semantics to this construction.

If we assume that `b_calculus` is a body for `calculus`, then the body for `complex` is:

```
body b_complex for complex;
  superimpose b_clock over b_calculus;
end;
```

Lastly, the `modvar`[6] is replaced by an array of `complex`, and any reference to `b_calculus` is replaced by `b_complex`. Every reference to an interaction point of `calculus` is replaced by the corresponding one of `complex`, in the initialize part of the surrounding *body* or *specification*.

[5] In opposition with *internal choice*.

[6] A `modvar` is an instance variable of a `module`.

4.2 The particular snapshots

4.2.1 The algorithm

A complete description of this algorithm can be found in [7]. A particular snapshot is a snapshot with empty channels. Therefore the state of the underlying system is given by the local states of all processes.

The algorithm is:

- Let us assume that there is a ring over the observers. A token is running on it. This token is a vector in Z^n, where n is the number of processes of the underlying calculus.
- Each observer has a vector of counters mt and behaves as follows:
  - When the site i sends a message to j the observer performs: $mt[j] := mt[j] + 1$
  - When the site i receives a message from j: $mt[i] := mt[i] - 1$
  - When the observer i receives the token, with the value *count*: It first waits until a *mark* is received on every input edge. Then it records the local state of the lower process. Then it computes $mt := mt + count$ and before sending the token with the value mt it sends a *mark* on every output edge. Lastly $mt := 0$.
  - When the token comes back to the master, if $count = 0$ then the master[7] broadcasts a message telling the observers to send him the recorded local states.

Theorem 2 (Partial correctness) *If the channels are FIFO, reliable and if we detect count = 0 at the end of the "tour" then the set of all the recorded local states is a particular snapshot.*

Theorem 3 (Total correctness) *If the underlying calculus accepts the inputs in any order and in a finite time then the token comes back in finite time*[8].

4.2.2 The program

There are two difficult points:

- How should we implement the snapshot of a local state? If we assume that the state of the underlying process is coded in a single variable (such as a record), then a single *peek* on that variable catches the local state.
- How should we implement the marks? The mark is simply a free message of the superimposed channel.

The channel type for the observers is:

[7]The initiator of the token, for instance process number 1.

[8]We assume that every message is delivered in a finite time.

```
channel ch_obs (in_o,out_o);
  by out_o : obs;
             mark;
end;
```

The module type of the observer processes is:

```
module megr(me:site);
  ip in_obs   : array[site] of ch_obs(in_o);
     out_obs  : array[site] of ch_obs(out_o);
     in_ring  : ch_ring(in_r);
     out_ring : ch_ring(out_r);
end;
```

And the complex is then:

```
module mcomplex(me:site);
  superimpose megr(me) over mcalculus(me);
  ip in_complex  : bind in_obs  over in_calculus;
     out_complex : bind out_obs over out_calculus;
     in_ring     : up in_ring;
     out_ring    : up out_ring;
  compose in_obs.obs  over in_calculus.info;
  compose out_obs.obs over out_calculus.info;
end;

body complex for mcomplex;
  superimpose egr over calculus;
  peek snap_local_state := s;
end;
```

In the observer the statement `rls := snap_local_state` performs an atomic copy of the state of the underlying process (assuming that the state is coded in `s`).

The interaction points `in_ring` and `out_ring` are implicit free interaction points of the module `mcomplex`. The message type `mark` is free, therefore not synchronized with any event

of the underlying process. Since the channels are FIFO, a composed message[9] (`obs , info`) can't overtake a mark.

The body of the observer is quite obvious and is not detailed here; it is using external choice on outputs.

5 The transformation method

The principle of the implementation of superimposition is to replace each superimposed body or module by a pure Estelle body which is semantically an implementation of the original construction.

The problem of the correctness of the implementation is raised in 5.5.

This algorithm takes as input *normal form* superimposed Estelle. This is why it is necessary to put the program in normal form.

5.1 The initial renaming

Before any computation, every symbol of the *specification* is renamed into a unique symbol:

- If the symbol is a field of a record then no renaming occurs. There cannot be any clash between two fields of two records.
- Otherwise it is sufficient to concatenate at the end of the symbol the unique number of the declaration environment.

There is also some renaming during the composition of two channels, modules, or bodies : For all free internally declared symbol, we append to them a tag in order to avoid clashes between symbols coming from the upper and lower channels, modules or bodies[10].

5.2 The composition of two modules

The algorithm for rewriting a superimposed module is:

Algorithm 1 (Superimposition of two modules) *Let us assume that the two modules are m_u and m_d.*

- *Every free interaction point of m_u and m_d is copied and renamed into the new module.*
- *For every composed interaction point, we must compute its channel type. Let us assume that the upper interaction point has a channel type c_u with a role r_u. Also for the lower interaction point : c_d and r_d are respectively its channel type and its role.*

  Such a channel is uniquely identified by the key:

  - *the upper channel type: c_u*

[9] We assume that `info` is the only message type of the underlying calculus. If there were several messages it would be sufficient to compose each of them with `obs`.

[10] Actually we append a "u" to the upper, free, locally defined symbols, and a "d" to the lower one.

- *the lower channel type:* c_d
- *the set of pairs of roles*[11]*:* $R = \{(r_u, r_d), (\overline{r_u}, \overline{r_d})\}$
- *the set of couples of composed messages:* $\{(m_u, m_d), \ldots\}$

If there exists an added channel with the same key, then the channel type of the composed `ip` *is this channel. Otherwise this is a new channel type. We must add it in the* `channel` *field of the embodying* specification *or* body.

- *The parameter list of the new module is the parameter list of the original composed module.*

We need to declare new channels. The process is:

- *the name is a fresh, unique symbol.*
- *for each pair of roles of the key, a new role is assigned. Let us denote* ρ *this mapping.*
- *for each pair of composed messages* (m_u, m_d)*, a new message is generated. Its roles are the roles corresponding to the pairs of roles of the two messages:* $\rho((\mathcal{R}(m_u) \times \mathcal{R}(m_d)) \cap R)$ *where* $\mathcal{R}(m)$ *is the set of roles of the message* m*.*
- *The parameter field is the concatenation of the two corresponding fields, with a renaming mapping* μ*, in order to avoid clashes.*

An example is given figures 2 and 3.

5.3 The Normal Form

Actually there are two normal forms. The first one consists in a replication process. The second is a transformation similar to the first normal form of [2].

5.3.1 The first normal form

The first normal form can be defined as follows:

Definition 1 (First Normal Form) *A* body *is in first normal form if and only if:*

- *At least one state is declared.*
- *The* `initialize` *statement exists, and has a* `to` *clause.*
- *Every transition has non empty* `from` *and* `to` *clauses.*

This transformation applies to the code of the upper and lower bodies of a superimposed body. Both must be terminal, not superimposed bodies.

[11] If r is the only role of a channel, then $\overline{r} = r$.

Algorithm 2 (First Normal Form) *The body must be a terminal body. It is rewritten as follows:*

- *if no explicit state is declared, then a* `state` *statement is added, with one new and unique state inside.*
- *If no* `initialize` *statement is declared, then an empty one is generated.*
- *If no* `to` *clause is specified in the* `initialize` *statement, then the automaton must have only one state, and a* `to` *clause with this state is added.*
- *for each transition:*
  - *If there isn't any* `from` *clause then one is added, the* `from` *states are all the states of the automaton.*
  - *If no* `to` *state is specified then the transition is replicated by the number of* `from` *states, with each of the* `from` *states in each* `to` *clause.*

Therefore the transitions have explicit *from* and *to* states.The correctness of this transformation comes directly from the Estelle standard [1].

5.3.2 The second normal form

Let Σ be a set of interactions (*output* or *when*)

Definition 2 (Second Normal Form) *A terminal body is in second normal form relatively to* Σ *if and only if:*

- *It is in first normal form.*
- *Every transition is in one of the 3 forms:*

  Boolean Form *It contains no interaction of* Σ.

  When Form *The transition contains one* `when` *clause on an interaction in* Σ. *And the statement part of the transition doesn't contain any event of* Σ.

  Output Form *The transition contains no* `when` *clause. The* **first** *statement of the statement part of the transition is an* `output` *event of* Σ. *The remainder of the statement part doesn't contain any event of* Σ.

The semantics of this *output form* is quite different from the Estelle semantics: the transition is fired only when the output is possible. Therefore the choice is external, which is required for observation algorithms. In VEDA [8] the observation of an event is performed by a `when` transition which has an external choice semantics. For reasons of simplicity of the implementation it has been chosen not to change the syntax of the transition, yet it would have been better to put the *external choice output* in the clause part of the transition.

The principle of the transformation from first normal form to second normal form is to recursively replace the body of each transition which is not in second normal form by an

automaton whose transitions are Estelle transitions (such an automaton has an initial and a final state), and then to flatten this representation in a single automaton, with second normal form transitions.

This algorithm requires an unbounded set of fresh variables and states. It also involves that some declarations are moved into the scope of the body. Figure 4 and 5 gives the automaton associated to each Estelle constructions.

The transformation for the "`with` $(\epsilon_j)_{j\in J}$ `do` S" construction is a particularly complex one: the `initialize` transition contains the assignment of each array index of $(\epsilon_j)_{j\in J}$ into a fresh variable. The second transition (see figure 4) contains the statement S in which every reference to a field f of one of the records $(\epsilon_j)_{j\in J}$ is replaced by the expression $\epsilon_j.f$, where the array indexes have been substituted by the corresponding fresh variables of the `initialize` statement.

The `goto` statement is forbidden.

Algorithm 3 (Flattening) *Every transition that is not in normal form is replaced by its automaton :*

- *the* `const`, `type`, `var`, `subroutine`, `state` *declarations are copied into the corresponding fields of the embodying automaton.*
- *the* `initialize` *transition is rewritten in the transition where:*
  - *The* `priority`, `from`, `when` *and* `any` *clauses are equal to the corresponding clauses of the including transition.*
  - *The other clauses remain untouched.*
- *The final state is discarded and the* `to` *clause of every transition going to this state is set to the value of the* `to` *clause of the including transition.*

Algorithm 4 (Second Normal Form) *For each transition not already in second normal form:*

- *If it is a transition with a* `when` *statement ω and some bounded* `outputs` *in the statement part $S_1;\ldots;S_n$. Then the transition is split into two transitions, placed in sequence. We assume that S_k $k \in \{1,\ldots,n\}$ is the first statement containing a bounded* `output`.
  1. *The first one is a transition containing ω, and all the statements $S_1;\ldots;S_{k-1}$.*
  2. *The second transition just contains the statements $S_k;\ldots;S_n$. It must be put in second normal form.*
- *If it is a transition with no* `when` *and only one bounded* `output`, *which is in the first statement (yet not in normal form). Then the Flattening algorithm is applied to this transition.*
- *If it is a transition with no* `when` *and a statement part of the form $S_1;\cdots;S_n$, where each S_i contains exactly one bounded* `output` *statement. Then this transition is split into* n *transitions, in sequence, with a S_i in each of them. They must be put in normal form.*

An example is given in the appendix 9.1.

5.4 The composition of two normal form bodies

This composition is the product of the two automata, where the bounded pairs of interactions are synchronized. If K is the binding mapping, then this composition takes as input:

- A domain(K)–second normal form body for the upper body.
- A range(K)–second normal form body for the lower one.

Algorithm 5 (Composition of Bodies) *the superimposed body is transformed into a terminal body with:*

- *The set Θ of states of the new body is isomorphic to the product of the two set of states Θ_u and Θ_d. Let us call σ the isomorphism: $\sigma : \Theta_u \times \Theta_d \longrightarrow \Theta$*
- *The constants, types, variables and subroutines declarations are the concatenation of the two corresponding fields, with the constraints:*
  - *The declaration of every variable declared as a peek of an underlying variable is discarded. Every occurence of this variable in the code is replaced by the corresponding underlying one.*
  - *The formal parameters of the two modules are declared as variables.*
- *The* `initialize` *transition is:*
  1. *A sequence of assignments on the parameters of the two constitutives modules, in order to share the parameters of the composed module.*
  2. *The* `initialize` *of the upper body.*
  3. *The* `initialize` *of the lower body.*

  The initial state is $\sigma(t_u, t_d)$ where t_u and t_d are the corresponding initial states.
- *Every* **boolean form** *transition of the upper body is replicated by the number of states of the lower body. Each time, for a state $t \in \Theta_d$, the* `from` *clause becomes $\sigma(from_u \times \{t\})$, and the* `to` *clause becomes $\sigma(t_u, t)$.*
- *Identically, for a* **boolean form** *lower transition. The* `from` *clause becomes $\sigma(\{t\} \times from_d)$, the* `to` *clause becomes $\sigma(t, t_d)$, for each $t \in \Theta_u$.*
- *And for every pair of matching bounded transition:*
  - *The* `priority` *clause becomes the sum*[12] *of the two clauses (if they exist).*
  - *The* `any` *clause is the concatenation of the two clauses.*
  - *The* `provided` *is the "*`and`*" of the two clauses, and of the equality tests of the matching indexes (if we compose an array of interaction points).*

[12]Yet the priority between two transitions of two distinct processes has no meaning, we chose this formula because it has a quite "natural" behaviour.

- *If the transitions are in* **when form**: *the* **when** *clause is the corresponding interaction.*
- *The* **from** *clause is the product of the two clauses:* $\sigma(from_u \times from_d)$
- *The* **to** *state is:* $\sigma(to_u, to_d)$
- *If the transition is in* **output form**, *the* **output** *statements are composed. The rest of the statement part are concatenated. Otherwise, the statement part are simply concatenated.*

An example is given appendix 9.2.

5.5 What about the correctness?

In this section, only the principles of the proof are given. This proof consists of two parts: The first one is the proof of the correctness of the normal form transformation. The second one is the proof of the correctness of the implementation of the superimposition.

5.5.1 Correctness of the transformation in normal form

The normal form transformation rewrites each process. We want to prove that the normal form specification has the same behavior. This is why we need a semantics $\cong$ of Estelle, which is a congruence.

If P and Q are two processes and if $A[.]$ denotes a context (a specification with a hole), then if $\cong$ is a congruence:

$$P \cong Q \implies \forall A[.]\ A[P] \cong A[Q]$$

Therefore it is sufficient to prove that the meaning of each process is not changed by the normal form transformation.

A failure semantics seems adequate to that purpose.

5.5.2 Correctness of the superimposition

Let us assume that:

1. There are no `priority` or `delay` statements in the specification.
2. The fairness assumption between the upper and the lower process is the same than the fairness of the transitions in a single process.

We assume that P denotes the original specification, P' the generated one, and $\mathcal{E}(P)$ the set of all computations of P. If s is one of these, then s_i is its projection on the process i.

Conjecture 1 (Partial Correctness) *Every computation of P' is a computation of P.*

Conjecture 2 (Total correctness) *Every maximal computation of P' is a maximal computation of P.*

We cannot go any further since (as it has been shown in [3]) it is possible to find a superimposed specification with a computation s which has the property[13]:

$$\forall s' \in \mathcal{E}(P') \ \exists i \ \ s_i \neq s'_i$$

These correctness proofs are absolutely tedious since Estelle is a complex language, with a complex informal semantics. As far as we know, giving a denotational semantics to Estelle has not been done yet.

5.6 The complexity of the generated code

Roughly speaking, if the number of transitions of each normal form body is n, and the number of states is m, the generated code for a superimposed body has $O(n^2) + O(m)$ transitions. The size of the code generated by the normal form transformation is in $O(n) + O(m)$. Thus the complexity is quadratic in the size of the code, and in the number of states of each body.

The generated code can become quite big if the number of states is high, or if there is a large number of matching interaction couples (thus a high degree of nondeterminism).

The run time complexity of the code is not changed except for the number of clause evaluations. Let us consider a completely connected graph of n processes. If every process sends and receives one message on each edge, with the mecanism used in Echidna [9] the number of transition evaluations is $O(n^3)$ on the whole system. Now if we superimpose on it the Fidge and Mattern's clock, the complexity is $O(n^4)$.

Therefore one must always bear in mind these complexity problems.

6 The implementation of the compiler

The superimposition compiler has been implemented in CAML[14]. The parser is derived from the parser of the Echidna compiler [9], which is written in Pascal. The generated code can be compiled by the Echidna compiler and then run on several distributed machines.

The code of the appendix 9 has been generated by this implementation.

On input, the language is not fully the normalized static Estelle (the `delay` construction is not implemented, there must be some explicit parameters in a `when` clause). On output, there are no *class* keywords, the parameters of a `when` clause can be different than the formal parameters in the channel declaration, the variable of a `for` statement is not always locally defined. We are working on meeting the Estelle standard.

7 Conclusion

This implementation of superimposition shows that, first of all, it is possible to write Estelle-programs transformation systems in a few months (it took actually about 2 months), although the syntax of Estelle is quite big. Secondly it allowed us to experiment superimposition and

[13]Let us denote s_i the projection of the calculus s on the process i. It is computed by discarding in s every action that does not belong to the process i.

[14]CAML is a dialect of ML, developped by INRIA. It is a functional strongly typed polymorphic language.

some superimposed algorithms, therefore proving that it is a valuable program composition technique. Lastly, the use of a strongly typed language has been a great help, since most of the bugs were detected at compilation time.

There are several interesting research directions on superimposition:

- A global (not distributed) superimposition of two *specifications* seems possible. It would enable a complete observation of a specification without changing a single line in it. The distributed superimposition would be an intermediate form.
- Integrating the superimposition in dynamic Estelle seems possible. It is just required that the processes and edges creations/destructions are treated as externally visible events, so that the superimposed process could be synchronized on it and perform a similar action, in order to maintain the equality of the networks.
- Optimizing the generated code is an important problem since it has been shown in section 5.6 that the size of the generated code is quite large.
- The correctness proof, although tedious, seems possible and is an important deal since the correctness of this compositional programming method is entirely grounded on this proof.

8 Acknowledgments

I would like to thank Claude Jard. He did much of the work of adapting the parser of the Echidna compiler [9] for the purpose of superimposition. His advices were always very relevant.

```
channel ch_obs(in_o,out_o);
  by out_o: obs;
  by out_o: mark;

channel ch_calculus(in_c,out_c);
  by out_c: info(i: vbool);

module megr(me: site; neighbours : neighbourhood);
  ip in_obs: array[cir] of ch_obs(in_o);
     out_obs: array[cir] of ch_obs(out_o);
     in_ring: ch_ring(in_a);
     out_ring: ch_ring(out_a);
  end;

module mcalculus(me: site; neighbours: neighbourhood);
  ip in_calculus: array[cir] of ch_calculus(in_c);
     out_calculus: array[cir] of ch_calculus(out_c);
  end;

module mcomplex(z:integer;me: site; neighbours: neighbourhood);
  superimpose megr(me,neighbours) over mcalculus(me,neighbours);
  ip in_complex: bind in_obs over in_calculus;
     out_complex: bind out_obs over out_calculus;
     in_ring: up in_ring;
     out_ring: up out_ring;
  compose in_obs.obs over in_calculus.info;
  compose out_obs.obs over out_calculus.info;
end;
```

Figure 2: An example of module composition: parts of the source.

```
channel Unique_1_U(Unique_2_U,Unique_3_U);
  by Unique_2_U : Unique_4_U(I_42Kd:Vbool_3K);
  by Unique_2_U : Mark_37Ku;

module Mcomplex_3K(Z_95K:Integer; Me_95K:Site_3K;
                   Neighbours_95K:Neighbourhood_3K);
  ip In_ring_95K:Ch_ring_3K(In_a_38K);
     Out_ring_95K:Ch_ring_3K(Out_a_38K);
     In_complex_95K:array [Cir_3K] of Unique_1_U(Unique_3_U);
     Out_complex_95K:array [Cir_3K] of Unique_1_U(Unique_2_U);
end;
```

Figure 3: An example of module composition: parts of the generated code

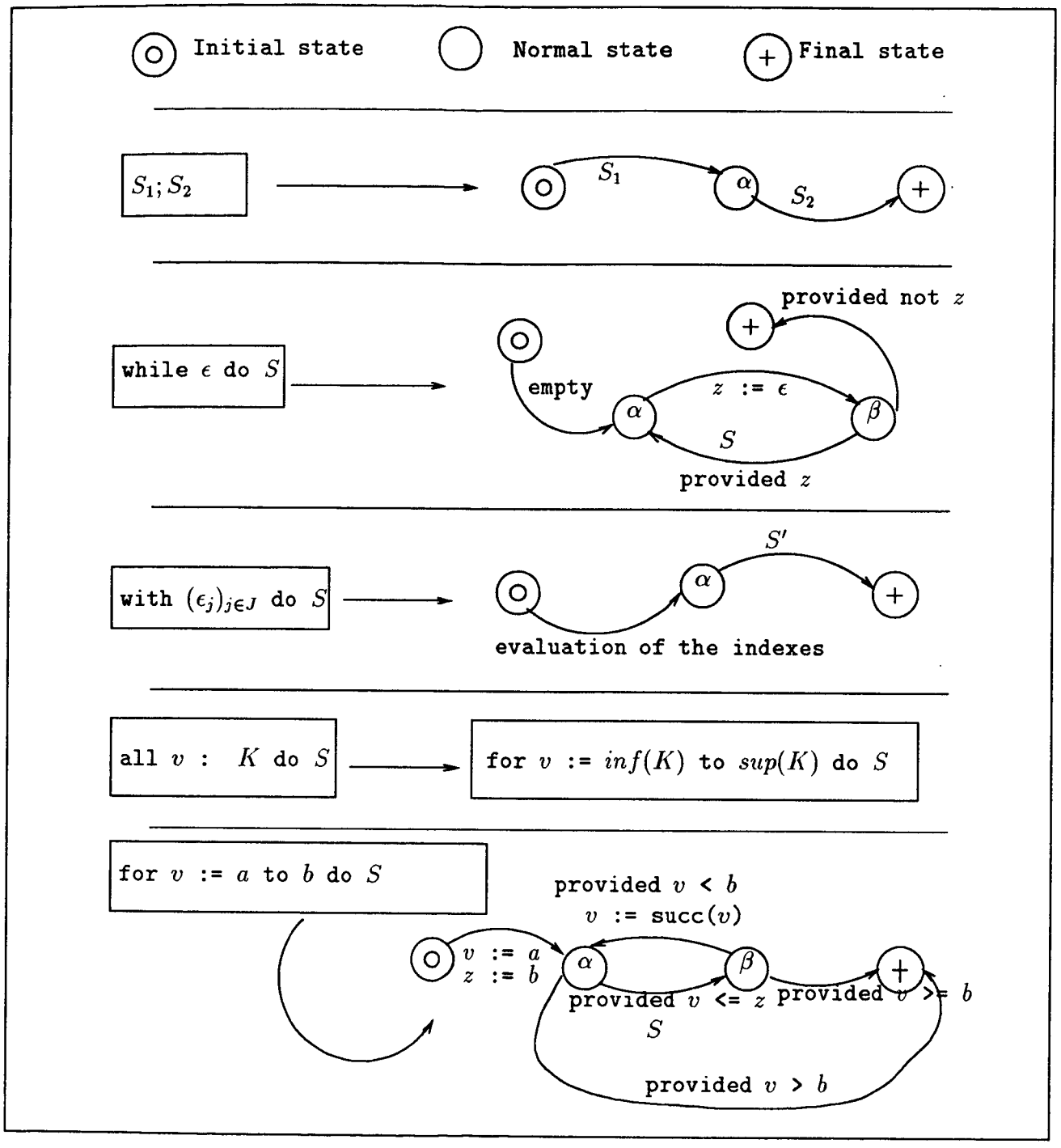

Figure 4: The automata associated to each Estelle constructions. Note that the only transition coming from the initial state is the `initialize` transition (Part 1).

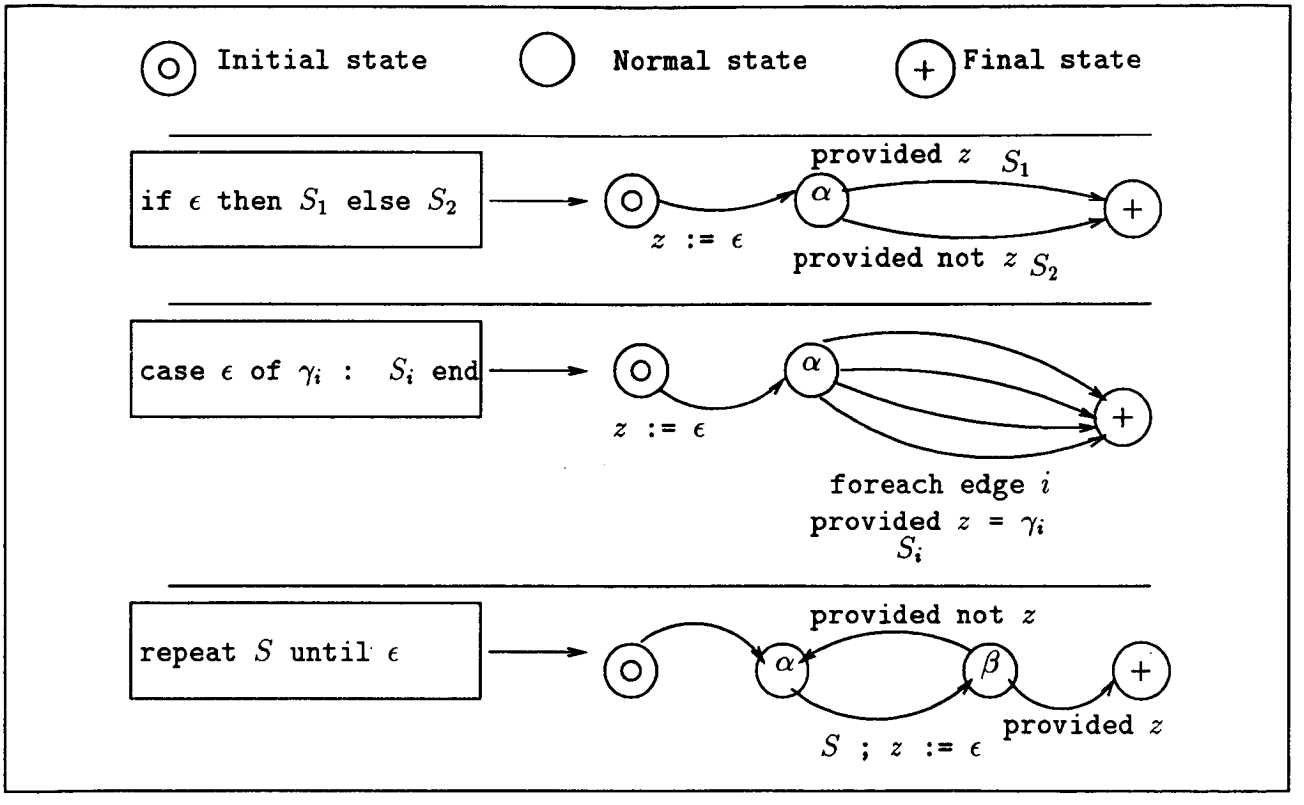

Figure 5: The automata associated to each Estelle constructions. Note that the only transition coming from the initial state is the `initialize` transition (Part 2).

9 Appendix

9.1 Example of normal form transformation of a body

The normal form transformation applied on the following:

```
specification fn_ex;

const k_max = 100;

type k = 1..k_max;

channel ch(i_ch,o_ch);
  by o_ch: msg;

module a;
  ip a1 : ch(o_ch);
end;

module b;
  ip b1 : ch(o_ch);
end;

body ba for a;

initialize
  begin
  end;

trans
  any x : k do
  begin
    if (1+1) = 3
      then output a1.msg
  end;

trans
  begin
    all y : k do
      output a1.msg
  end;
end;

body bb for b;

initialize
  begin
  end;

trans
  begin
    output b1.msg
  end;
end;

module c;
  superimpose a over b;
  ip c1 : bind a1 over b1;
  compose a1.msg over b1.msg;
end;

body bc for c;
  superimpose ba over bb;
end;

initialize
  begin
  end;

end.
```

and composed over a body with one transition, and a single bounded **output**, gives:

```
specification Fn_ex_2K;

const K_max_3K=100;

type K_3K=1..K_max_3K;

channel Unique_1_U
  (Unique_2_U,Unique_3_U);
 by Unique_2_U : Unique_4_U;

 { deleted code ..... }

module C_3K;
 ip C1_15K:Unique_1_U
           (Unique_2_U);
end;

body Bc_3K for C_3K;

var Unique_12_Uu:Boolean;
    Unique_10_Uu:K_3K;
    Unique_8_Uu:K_3K;
    Y_11Ku:K_3K;

state
  Unique_18_U,Unique_17_U,
  Unique_16_U,Unique_15_U,
  Unique_14_U;

initialize to Unique_14_U
  begin
  end;

trans  { A }
 from Unique_14_U
 to Unique_16_U
 any X_9Ku:K_3K do
 begin
  Unique_10_Uu:=X_9Ku
 end;

trans  { B }
 from Unique_16_U
 to Unique_15_U
 begin
  Unique_12_Uu:=(((1+1))=3)
 end;

trans  { C }
 from Unique_15_U
 to Unique_14_U
 provided (not Unique_12_Uu)
 begin
 end;

trans  { D }
 from Unique_14_U
 to Unique_18_U
 begin
  Y_11Ku:=1;
  Unique_8_Uu:=K_max_3K
 end;

trans  { E }
 from Unique_18_U
 to Unique_14_U
 provided
  (Y_11Ku>Unique_8_Uu)
 begin
 end;

trans  { F }
 from Unique_17_U
 to Unique_18_U
 provided
  (Y_11Ku<Unique_8_Uu)
 begin
  Y_11Ku:=Succ(Y_11Ku)
 end;

trans  { G }
 from Unique_17_U
 to Unique_14_U
 provided
  (Y_11Ku>=Unique_8_Uu)
 begin
 end;

trans  { H }
 from Unique_15_U
 to Unique_14_U
 provided Unique_12_Uu
 begin
   output
    C1_15K.Unique_4_U
end;

trans  { I }
 from Unique_18_U
 to Unique_17_U
 provided
  (Y_11Ku<=Unique_8_Uu)
 begin
  output
   C1_15K.Unique_4_U
 end;
end;

{ deleted code ..... }

end.
```

Figure 6 gives the automata associated to the source and the object bodies.

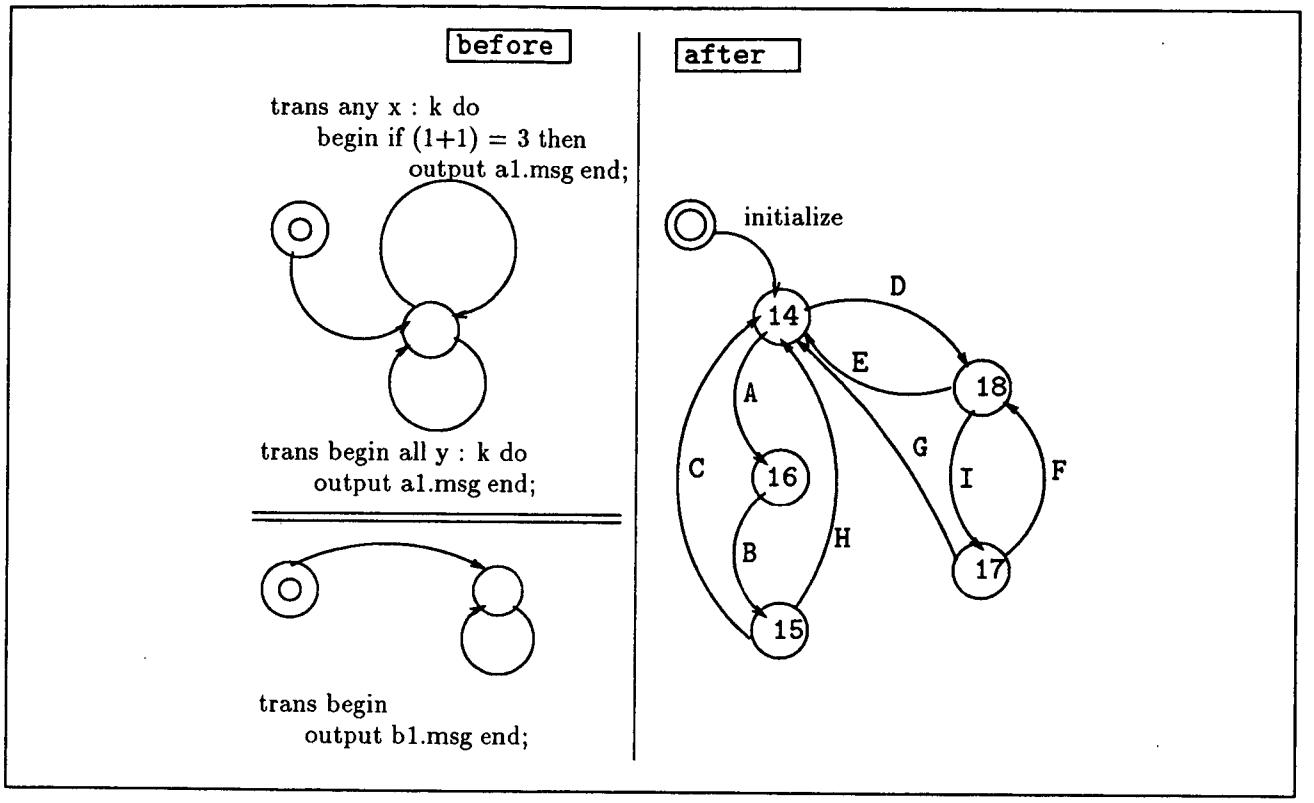

Figure 6: The associated automata

9.2 Example of composition of two normal form bodies

The source code is:

```
specification cp_ex;

const k_max   = 100;
      cpt_max = 1000;

type k = 1..k_max;

channel ch(i_ch,o_ch);
  by o_ch: msg1;
  by o_ch: msg2;

module a;
 ip a1: array[k] of ch(o_ch);
    a2: array[k] of ch(i_ch);
    a3: ch(o_ch);
end;

module b;
 ip b1: array[k] of ch(o_ch);
    b2: array[k] of ch(i_ch);
end;

body ba for a;

var pk : integer;
    m  : integer;

state alpha,beta;

initialize
  to alpha
  begin
  end;

trans
  from alpha
  to beta
  any x : k do
  begin
    output a1[x].msg1
  end;

trans
  from alpha
  to beta
  any x : k do
  begin
    output a1[x].msg2
  end;

trans
  from beta
  to alpha
  any x : k do
  when a2[x].msg1
  provided (x = 7)
  begin
    m := pk
  end;

trans
  from beta
  to alpha
  begin
    output a3.msg1
  end;
end;

body bb for b;

var z : integer;

initialize
  begin
    z := 0;
  end;

trans
  any x : k do
  begin
    output b1[x].msg1
  end;

trans
  any x : k do
  when b2[x].msg1
  provided (z < cpt_max)
  begin
    z := z + 1
  end;
end;

module c;
 superimpose a
        over b;
 ip c1: bind a1 over b1;
    c2: bind a2 over b2;
    c3: up a3;
 compose a1.msg1
    over b1.msg1;
 compose a2.msg1
    over b2.msg1;
end;

body bc for c;
 superimpose ba over bb;
 peek pk := z;
end;

initialize
  begin
  end;

end.
```

And the object code is:

```
specification Cp_ex_2K;

const K_max_3K=100;
      Cpt_max_3K=1000;

type K_3K=1..K_max_3K;

channel Unique_1_U(Unique_2_U,
                   Unique_3_U);
  by Unique_2_U : Unique_4_U;
  by Unique_2_U : Msg2_4Ku;
  by Unique_2_U : Msg2_4Kd;

{ deleted code ..... }

module C_3K;
  ip C3_17K:Ch_3K(O_ch_4K);
     C1_17K:array [K_3K] of
       Unique_1_U(Unique_2_U);
     C2_17K:array [K_3K] of
       Unique_1_U(Unique_3_U);
end;

{ deleted code ..... }

body Bc_3K for C_3K;
  var M_7Ku:Integer;
      Z_13Kd:Integer;

state Unique_7_U,Unique_6_U;

initialize
 to Unique_6_U
 begin
   Z_13Kd:=0;
  end;
```

```
trans
  from Unique_6_U
  to Unique_7_U
  any X_10Ku:K_3K do
  begin
    output C1_17K[X_10Ku].Msg2_4Ku
  end;

trans
  from Unique_7_U
  to Unique_6_U
  begin
    output C3_17K.Msg1_4K
  end;

trans
  from Unique_7_U
  to Unique_6_U
  any X_11Ku:K_3K; X_16Kd:K_3K do
  when C2_17K[X_11Ku].Unique_4_U
  provided (((X_11Ku=7)) and
            (((Z_13Kd<Cpt_max_3K))
             and (X_11Ku=X_16Kd)))
  begin
    M_7Ku:=Z_13Kd;
    Z_13Kd:=(Z_13Kd+1)
  end;

trans
  from Unique_6_U
  to Unique_7_U
  any X_9Ku:K_3K; X_15Kd:K_3K do
  provided (X_9Ku=X_15Kd)
  begin
    output C1_17K[X_9Ku].Unique_4_U
  end;
end;

initialize
  begin
  end;
end.
```

References

[1] IS 9074. *Estelle: a Formal Description Technique based on an Extented State Transition Model.* ISO TC97/SC21/WG6.1, 1989.

[2] K.R. Apt, L. Bougé, and P. Clermont. *Two Normal Form Theorems for CSP Programs.* Technical Report 10, LIENS, Ecole Normale Supérieure, Paris, France, June 1987.

[3] L. Bougé and N. Francez. A compositional approach to superimpostion. In *Proc. of the* 15^{th} *ACM SIGACT-SIGPLAN Symposium on Principle of Programming Languages*, pages 240–249, San Diego, California, January 1988.

[4] B. Caillaud. La superposition. Mémoire de D.E.A., Univ. Paris 6, Paris, France, Septembre 1988.

[5] K. M. Chandy and J. Misra. *Parallel program design : a foundation.* Addison-Wesley, 1988.

[6] N. Francez. *Fairness.* Springer Verlag, New York, 1986.

[7] J.M. Hélary, N. Plouzeau, and M. Raynal. A characterization of a particular class of distributed snapshots. In *Proc. International Conference on Computing and Information (ICCI'89), Toronto*, North–Holland, may 23–27 1989.

[8] C. Jard, R. Groz, and J.F. Monin. Development of VEDA: a prototyping tool for distributed algorithms. In *IEEE Trans. on Software Engin.*, March 1988.

[9] C. Jard and J.-M. Jézéquel. A multi-processor Estelle to *C* compiler to experiment distributed algorithms on parallel machines. In *Proc. of the* 9^{th} *IFIP International Workshop on Protocol Specification, Testing, and Verification, University of Twente, The Netherlands*, North Holland, 1989.

[10] F. Mattern. Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, and Robert, editors, *Proc. Int. Workshop on Parallel and Distributed Algorithms, Bonas, France, oct. 1988*, North Holland, 1989.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 486 **SYNTHESIS OF A NEW SYSTOLIC ARCHITECTURE FOR THE ALGEBRAIC PATH PROBLEM**
Abdelhamid BENAINI, Patrice QUINTON, Yves ROBERT, Yannick SAOUTER, Bernard TOURANCHEAU
34 Pages, Juillet 1989.

PI 487 **PLANS SIMULATION USING TEMPORAL LOGICS**
Eric RUTTEN, Lionel MARCE
40 Pages, Juillet 1989.

PI 488 **ON FINITE LOOPS IN LOGIC PROGRAMMING**
Philippe BESNARD
20 Pages, Septembre 1989.

PI 489 **LTA : UN LANGAGE DE TRAITEMENT D'ARBRES**
Dalila HATTAB
24 Pages, Septembre 1989.

PI 490 **THE SIGNAL SOFTWARE ENVIRONMENT FOR REAL-TIME SYSTEM SPECIFICATION, DESIGN, AND IMPLEMENTATION**
Albert BENVENISTE, Paul LE GUERNIC
34 Pages, Septembre 1989.

PI 491 **PHYSIQUE QUALITATIVE : PRESENTATION ET COMMENTAIRES**
Qinghua ZHANG
48 Pages, Septembre 1989.

PI 492 **SPARSE MATRIX MULTIPLICATION ON VECTOR COMPUTERS**
Jocelyne ERHEL
20 Pages, Septembre 1989.

PI 493 **THE SUPERIMPOSITION OF ESTELLE PROGRAMS : A TOOL FOR THE IMPLEMENTATION OF OBSERVATION AND CONTROL ALGORITHMS**
Benoît CAILLAUD
30 Pages, Septembre 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique