Some Estelle mechanisms revisited

Stanislaw Budkowski [1], Piotr Dembinski [2]

(1) Software-Networks Department, Institut National des Télécommunications, 9 rue Ch. Fourier, 91011Evry, France
tel: +33 (0)1 60 76 47 20
fax: +33 (0)1 60 76 47 11
e_mail: stan@int-evry.fr

(2) Institute of Computer Science, Polish Academy of Sciences,
Ordona 21, 01-237 Warasaw, Poland
tel: +48 22 36 28 41
fax: +48 22 376564
email: piotrd@wars.ipipan.waw.pl

I. Introduction

We try to make an (incomplete) outline of Estelle mechanisms that have been discussed over last years. We think it is time to summarise some of the experience and be prepared for broader discussion on the language futures. We repeat and remind some arguments that were raised while the language was being designed, add some new ones and give references to works that addressed a particular subject. The mechanisms that are discussed below are not ordered by any priorities, and many of them are strongly inter-related. The list is by no means complete. We try to avoid any conclusive remarks, but our preference is to keep the original principles and "logic" of the language. We are making, however, some concrete proposition related to inter-module communication and structuralization.

Also, it should be stressed that we opt for a clear separation of purely specification (language) problems from (very important) problems concerning implementation and its efficiency. The latter are not treated here.

Only some of the mechanisms outlined below will be discussed deeply.

(1) Module input queues (buffers)

There have been a number of remarks concerning Estelle asynchronous communication based on unbounded FIFO input queue. Possible enhancement consist in considering a more general concept , that is, input buffers. This concerns the buffer organization, the way an input buffer is accessed in read and write by the module the buffer belongs to, and the way the connected, external modules append interactions.

Currently, the Estelle module can only read the type (name) of the head-of-queue interaction in the *when*-part of an enabling conditions and the interaction parameters in the *provided*-part of it, it may read/write the parameters in the transition's action, and the action always removes the interaction from the queue.

In this section we are going to discuss propositions concerning only point-to-point asynchronous communication and preserving the basic assumptions of such communication in Estelle, such as:

- an input buffer is attached to (an) interaction point(s) of a module, hence the external modules may only append interactions to it (it may be discussed, however, what does exactly append mean)

- the module itself has a restricted access to its input buffers and interactions stored in them ; the way this restrictions are defined is the subject of discussion.

These propositions, possible solutions and their consequences for the language, have been discussed in a more general framwork in [10].

(a) <u>Relaxation of FIFO queue restriction</u>

Thinking of relaxation of FIFO queue restriction, one can imagine an extreme proposition which would say that input buffers are to be completely specified by the user in that he would be responsible for defining their (abstract) types by means of allowed operations and their properties (in the style of abstract data types in ACT ONE, LOTOS or SDL). Such an approach certainly does not propose a simple extension of the basic model and creates more problems than it solves therefore it is not discussed here.

A more practical approach of a similar nature may be proposed, i.e., user defined buffer types with data structures taken from (extended/restricted) Pascal and with accessible operations defined as (extended/restricted) Pascal procedures or functions. This approach seems attractive, but also requires a lot more definitions in the language and, what is important, makes the existing distinction between when-clause and provided-clause meaningless (although one can imagine that a predefined buffer type "queue" with "when" predicate/operation, in the present sense, may exist). On the other hand, such a generalized "provided-clause" has it good sides in that it may also eliminate (or solve) problems signalized in (c) below. However, it should be also remembered that one of the Estelle assumption was to separate, as far as possible, Pascal from purely Estelle constructs. This has been preserved to great extent and the proposed approach reintroduces the problem (Pascal functions and procedures addressing Estelle constructs - this is the subject of (9) below). It seems, however, that the approach is worth to be studied further.

<u>(b) Size (capacity) of the input buffer.</u>

Many propositions have been made to enable declaration of the size limit of the input buffer (queue). This enhancement is rather easy to introduce but requires a discussion on the treatment of the "appending" policy when a buffer is full (e.g., incoming message is lost or it overwrites the last one). Any solution is also closely depended on the one adopted for point (a) above.

<u>(c) Enabling condition depending on many interactions in possibly different input buffers.</u>

This postulate combines several sub-postulates concerning a possibility of checking whether:

- "the message of type m1 is in head of ip1-queue and the message of type m2 is in head of ip2-queue, i.e., when (ip1.m1 and ip2.m2), for ip1 <> ip2", or

- "the message of type m1 is in head of ip1-queue or the message of type m2 is in head of ip2-queue, i.e., when (ip1.m1 or ip2.m2)" (a solution to this postulate was proposed by R. Tenney in [8] by introducing an "interactionset" concept, similar to the "stateset" existing in Estelle), or

- "the message of type different than m1 is in head of ip1-queue, i.e., when ip1.(not m1)",or

- "the message of type m1 is not in head of ip1-queue", i.e., when (not ip1.m1)"; this being negation of "when ip1.m1" constitutes the alternative "(the message of type different than m1 is in head of ip1-queue) or (the queue is empty)")

Note that::

- in a natural way "when (ip1.m1 or ip2.m2)" can be treated as a short-hand notation denoting two input-transitions each one with a simple when-clause. This is exactly the same kind of a short-hand as it has been recognized for a from-clause used in form "from S1, S2". Note, however, that this from-clause (and, more generally, "from stateset" clause) cannot be treated as short-hand in presence of a delay-clause. Input transitions are not allowed to include delay-clauses, in present version of Estelle, but this enhancement has been also postulated (see (6)). The conclusion is that "when (ip1.m1 or ip2.m2)" could not be treated as short-hand in presence of a delay-clause. In any case, the interpretation of "when (ip1.m1 and ip2.m2)" cannot be reduced to a short-hand.

- "when ip1.(not m1)" can be replaced by "when (ip1.m2 or ip1.m3 or... or ip1.mk)", where m2,...,mk are all interactions that may be received by ip1 but the interaction m1. Therefore it also can be treated as a short-hand notation.

- applying general boolean low "not (A or B) = not A and not B), "when (not ip1. m1)" could be replaced by "when (ip1.m2 or ip1.m3 or... or ip1.mk) provided (not is_empty(ip1))" , if the predicate "is_empty" was accessible in Estelle. Hence, "when (not ip1. m1)" equals "when ip1. (not m1) provided (not is_empty(ip1)".

- in "when (ip1.m1 or ip2.m2)" ip1 may equal ip2, while in "when (ip1.m1 and ip1.m2)" it has no sense (the logical value is always false) as long as the test concerns only the head of the queue.

- using "when (ip1.m1 and ip2.m2)" one confronts the problem of interaction parameters of the same identifiers. They can be used in provided-clause or within the transition's action and this may lead to a confusion. Therefore, a minimal (statically recognized) condition should be postulated, namely: every parameter identifier appearing within the scope of a when-clause must belong to a unique interaction among those interactions that occur in this when-clause. The word "occur" means rather a set-theoretical membership relation than simple occurrence of an identifier in the text, since such clauses like "when (ip1.not m1 and ip2.not m2)" are now admissible.

- another problem with "when (ip1.m1 and ip2.m2)" in a transition is such that it is not clear what happens to interactions m1 and m2 if the action of the transition is executed. Is one, both or none of them removed? To follow the existing Estelle rules, it seems most natural to remove both of them.

(d) Allowing dynamic creation of interaction points.

It seems possible to dynamically create interaction points, in the similar way as their connections (attachments), without loosing the possibility of static type checking. Such a solution could also satisfy those who preferred to have more complicated than arrays data structures of interaction points.

(e) Parallel channels (of signals or events)

This type of channels, useful in some low-level applications, is difficult to express in Estelle because it does not queue incoming signals. An input buffer for a parallel channel can be imagined as a collection of signal-holders of capacity 1 - one signal-holder for each signal. A consecutive signal overwrites its predecessor (see (b) above).

It is not worth, for this single case, to make an extension of the buffer notion. We think that with the possibility of introducing buffers (queues) of finite capacity (see (b) and the detailed proposition in the end of this section) and playing cleverly with names one can achieve the intended description without to much of an overhead. An example of module interfaces communicating through such a "parallel channel", is given below:

```
channel parallel_channel (A, B);
 by A: a1, a2, ... , ak;
 by B: b1, b2, ... , bm;

signal_type = ( a1, a2, ... , ak,b1, b2, ... , bm );

module sender activity;
 ip p : array[signal_type] of parallel_channel(A) individual queue(1)
end;

module receiver activity;
 ip p : array[signal_type] of parallel_channel(B) individual queue(1)
end;

modvar send: sender; rec: receiver;
•
(*******connection*******)
all x: signal_type do connect send.p[x] to rec.p[x];
•
(*******the only allowed form of output*******)
output p[x].x

(*******where x is admissible identifier in signal_type*******)
```

(f) Extending tests on buffers in provided clause

This postulate overlaps some other propositions concerning points which have been discussed above. In the exemplary extension described in details below we added one primitive function (buffer length) with which many useful tests can be produced (but, in general, we would hesitate to promote extensions by simple addition of predefined primitives).

Proposed extension

The proposed extension may be summarized as follows:

Adding mailbox-like input buffers, introducing buffers of finite capacity, extending when-clause, adding "length" test

Trying to preserve as much as possible the existing style and interpretation of Estelle and, on the other hand, trying to satisfy the postulates (a)-(c), one may first note that the organization of the buffer is not important (as well as its implementation). What really counts are user-visible and user-accessible testing predicates and operations manipulating the buffer. With this in mind, we will first try to extend Estelle allowing for both queue and mailbox-like buffering mechanisms, still however viewing a buffer as an ordered sequence of messages of bounded or, potentially, infinite length. The two types of buffers will be distinguished by key words "queue" and "mbox". Then, for these two types, when-clause is going to be extended in the style described in (c).

In this model, we need not change the semantics of writing a message (interaction) to the buffer by an external, connected module (output-statement). The message is simply appended to the buffer, as it was before, provided there is still place in it. Otherwise, the last element in the buffer is overwritten (alternatively, the newly arriving interaction is lost). The whole differentiation between queue and mailbox is done by different treatment of the when-clause. In case of a queue the interpretation of "when ip.m" remains the same as it is presently, while for a mailbox

> "when ip.m" is true iff "there exist message of type m in the ip-buffer" (i.e., therefore, the mbox is not empty)

The problem however arises when there is more than one interaction of the same type in the mailbox. Which one is to be tested if the interaction parameters appear in the following provided-clause or/and which one has to be removed when the transition is going to be executed? The first (the oldest) or randomly selected one from among those of type m? What happens if the test fails to be true for this selected one but there is another satisfying appropriate conditions? Each solution seems to have bad sides.

Choosing the first, it has this advantage that it doesn't create nondeterminism and keeps *when*- and *provided* clauses separated, but it has the inconvenience of not exactly corresponding to what a mailbox.was expected to be. (a sort of a *multi-queue* is rather considered, i.e., a common box of separate queues, one for each type of message).

Recognizing the set of all interactions of actual parameters satisfying conditions of the provided clause and then choosing one of them while executing seems the closest to the mailbox intuition, but creates a nondeterminism difficult to control. Besides, the whole combination resembles rather a "forone....suchthat....do" statement and not what originally had been expected. Choosing always the first (the oldest) satisfying the provided-clause, removes the non-determinism but still mixes up when- and provided-clause.

Choosing randomly one interaction of type m in the buffer, and then check the provided condition, seems to have only bad sides of the both preceding solutions and no positives.

Let us decide to take the first solution i.e.,multi-queue interpretation of the mailbox access. With all the above assumptions we replace Estelle notion, known as "queue-discipline" (defined as "common queue" or "individual queue"), with:

buffer-discipline = discipline buffer-type ["(" buffer-capacity ")"]
discipline = "common" | "individual"
buffer-type = "queue" | "mbox"
buffer-capacity = positive-integer

If buffer-capacity is omitted, then a potentially infinite buffer is assumed. Common buffers are only possible for interaction points of the same type.

Also, recall an assumption saying, that an interaction sent into a full buffer overwrites the last one in it ,i.e., the last regardless the type of it.

Let us try now to extend the when-clause to satisfy the needs explained in (c). The language of expressions admissible for a when-clause, is constructed in such a way that it gives a maximal expressive power but has a constrained syntax (expressions can only be written in, so called, additive normal form). This form also helps to understand what is only a short-hand notation and what a real Estelle extension. The Estelle syntax is extended as follows:

```
when-clause = "when" ip-expression
ip-expression =         or-ip-expression |
                        and-ip-expression |
                        "(" ip-expression ")"
or-ip-expression = and-ip-expression "or" and-ip-expression
and-ip-expression = simple-ip-expression |
                        simple-ip-expression "and" and-ip-expression
simple-ip-expression =
                        when-ip-reference "." interaction-identifier
                                              [ intaraction-argument-list ] |
                        when-ip-reference "." "not" interaction-identifier
                                              [ intaraction-argument-list ] |
                        when-ip-reference "." "anyone"
```

Examples:

when p.m,

when p.not m,

when (p1.m1 and p2.not m2) or p3.m3

The extended clause is interpreted exactly as it was explained in (c), with "or" operators used as short-hands (remember that it is only possible for input transitions without a delay-clause).

Additionally, we assumed that, in case of type "mbox" buffer, if when clause is satisfied for a simple-ip-expression, then the interaction taken into account within the scope of the clause, is the first one of those satisfying the clause. This is also the interaction removed in execution (and the tailing interactions pushed ahead to fill the gap, if necessary, in the buffer).

If *and-ip-expression* of a *when*-clause is satisfied, then all those interactions satisfying *simple-ip-expressions* of its components are taken into account. Their parameters appearing in the scope are assumed to be disjoint. All these interactions are then removed in execution of the transition action.

"anyone" is a new reserved word and:

"*when p.anyone*" is semantically equivalent to "when p.not m or p.m", for any m which can be received by p

In other words, ."*when p.anyone*" is true iff the head interaction in the buffer is anyone such that it may be received by p, for buffer-type being "queue", or there is an interaction acceptable by p in the buffer, for buffer-type being "mbox" (thus "*when p.anyone*" is another short-hand).

Because *simple-ip-expressions* "p.not m" and "p.anyone" are short-hands, and so is the use of "or" construct, therefore each transition with a when-clause represents a number of input transitions, each one of them having a when-clause with only one and-ip-expression which contains neither "p.not m" nor "p.anyone" simple-ip-expressions. This is then the form of when-clause which may appear in a simple transition (in terms used in Estelle definition) and it is the subject of semantic interpretation. To illustrate this point let us take an example.

```
trans
  when (iq.not q1 and ir.r1) or (is.not s1)
    begin stm end
```

Assume that the set of interactions that can be received by iq is {q1, q2, q3} and this received by is {s1, s2, s3}. The above transition expands first into

```
trans
  when (iq.not q1 and ir.r1)
    begin stm end

trans
  when (is.not s1)
    begin stm end
```

Because "iq.not q1" is equivalent to "iq.q2 or iq.q3", "is.not s1" is equivalent to "is.s2 or is.s3" , and (A or B) and C means the same as (A and C) or (B and C), then the final expantion results in

```
trans
  when (iq.q2 and ir.r1)
    begin stm end

trans
  when (iq.q3 and ir.r1)
    begin stm end

trans
  when is.s2
    begin stm end

trans
  when is.s3
    begin stm end
```

Finally, this proposition adds predefined function "length(ip)" returning the current number of interactions in the ip-queue (a variant of this would also allow to have a function "length(ip, m)" returning the current number of interactions of type m in the ip-queue). With such a simple function many useful tests on the buffer content can be defined, e.g., "buffer is empty" - length(ip)=0, buffer is full (only for buffers of finite capacity n) - length(ip)=n, "there is n messages (no message, if n=0) of type m in the buffer" - length(ip,m)=n, etc.

Thus the syntax and semantics of present Estelle is preserved. There is, however, a number of places in the Estelle definition, apart from the mentioned above, in which appropriate changes are to be inserted.

(2) Extension of message exchange communication (multicasting)

Estelle is based on point-to-point communication. A form of multicast (broadcast) can be only programmed, within a transition, by a multiple outputs through separate *ips* connecting modules. It seems insufficient. The problem is discussed in detail in [3] and very general solution is proposed there. The complexity of this solution is related to Estelle hierarchical structure and dynamism. Common queues add to this complexity, or in other words and to some extent, the problem of common versus individual queues is dual to that concerning the point-to-point versus multi-point communication. Any solution is strongly dependent on (1) and (7).

(3) Inter module communication mechanisms (shared, imported, exported variables, etc.)

Communication mechanisms other than message passing were very seriously discussed and finally rejected (with exception of restricted child-to-parent exported variables) on the basis that they create too many semantical difficulties (see SDL problems with these notions). It should be stressed, however, that implementations, where possible, can realise the basic communication mechanism efficiently (eg. in monoprocess environment) by shared variables. Restricted variables sharing between sibling modules of the "activity" parent module may be considered.

(4) Structuralization and parallelism

Currently, the Estelle module structuralization is asymetric w.r.t. "process" and "activity" (modules attributed with **activity** or **systemactivity** may be substructured *only* into modules attributed with **activity** *and not process*). An extension in the form that is fully compatible with the existing standard has been ready since 1987 [4]. In this proposition the only module nesting principle for defining modules within modules is:

(a) Every active module must be attributed (i.e., if for a given ***module-header-definition*** *there is at least one* ***module-body-definition*** *with its transition-part nonempty, then the* ***module-header-definition*** *must include one of the two key-words:* ***process*** *or* ***activity****).*

Please observe that inactive module may be attributed.

The above principle results in a natural way from the property of the parent/children priority. This property requires certain global control over executions of the children of an active instance but does not determine the character of this control. The attributes **process** and **activity** serve to distinguish two possible ways of controlling executions of the module children. These two possibilities are:

(1) *Synchronous* parallel execution of children instances in case of a process parent. This means that, at a given moment, either the parent executes or (if the parent instance has nothing to execute) all its children in parallel.

(2) *Nondeterministic* execution among children instances in case of an activity parent (always preserving the parent/children priority). This means that at given moment either the parent executes or one of its children. Another consequence of the attributing principle (a) is that an attributed module may be active or inactive. Thus even an inactive module instance may force a required synchroni- zation among its children if appropriately attributed.

A framework module instance (instance of an inactive and unattributed module) does not assume direct global control over its children. The control may, however, come from an upper level instance as stated by the property (3) below.

(3) A framework module instance which is a descendant of an attributed instance constraints the behavior of its children instances in the way indicated by the attribute of its closest attributted ancestor. In other words, an unattributed module **inherits** the attribute of its closest embodying and attributed module.

The following is a direct consequence of (3):

(4) The asynchronous independent behavior among module instances is only possible if they all are instances of modules which are not embeded in an attributed module (all embedying modules are inactive and unattributed i.e., they are framework modules).

From the principle (a) there is exactly one attributed module, along any path within the hierarchy of modules within a specification, for which all embodying modules are inactive framework modules (except in the degenerate case, where all specification modules are inactive and consequently may remain unattributed). Consequently, any initial state of a specification defines a fixed number of instances these modules and interaction point links above them. This structure once initialized cannot change during the life-time of the specified configuration since parent modules are inactive; it is invariant. Such a top-level attributed module (and each of its instances) is often called **system module** (module instance) and, respectively to the module attribute, either **systemprocess** or **systemactivity**. A specification itself will reduce to a single system module if it is attributed (and it must be attributed if it is active).

We propose to keep a syntactical facility in Estelle allowing qualification of such a system module within a specification with the attribute (keyword) **systemprocess** or **systemactivity**.

By the property (4) above systems and only systems behave fully **asynchronously** with respect to each other. The use of the system attributes clearly distinguishes this level of structuring within a specification.

Please note that there is no changes in the definition of AS(gidp) wrt the ISO standard.

<u>Other remarks</u>

Some user prefer static and "flat" Estelle (eg. Veda [6]) because it avoids many problems. On the other hand, the dynamic facilities of Estelle seems to be attractive.

Its also true that "process" synchronous parallelism is rarely (and, what much worse, unproperly) used. However, some interesting applications of that type of parallelism can be found, for example when one want to achieve the effect similar to (semantics of) synchronous languages as e.g. Esterel. Making structuralization completely symmetric (see above) may increase the importance of synchronous parallelism.

Other complains w.r.t Estelle parallelism concern the differentiation between asynchronism and interleaving since the difference may really occur only in presence of priorities (either parent-child and transition priorities). This is true in present Estelle - it is not, if some extensions considered in (1) were adopted. In this context priorities are also contested. Our experience is that existence of priorities is beneficial in some situation if message exchange communication is the only available mechanism. Therefore, the "priority" discussion must be related to (8).

Some discussion about the need to increase the concurency in Estelle is also appropriate (see [11].

(5) Module internal automata and "nextstate" construct

This problem has been permanently in discussion since the beginning. The current solution (without "nextstate" statement) was chosen on the basis that the statement destroys the clean automata-like definition of the module behaviour. Additional argument was that the statement may be totally emulated, if needed, by the construct illustrated below:

```
type STAT = [S1,..,SN]        {enumerated type};
stat_var : STAT;
:
trans
provided stat_var=Si
begin
:
if p then stat_var:=Sj else stat_var:=Sk          {nextstates}
:
end;
```

Of course, explicit "nextstate" statement has some advantages over the solution above and the discussion is appropriate.

(6) Delay clause for input transitions

In the current Estelle definition the presence of a **when**-clause excludes a **delay**-clause and vice versa. This restriction may be dropped, but the discussion about its importance is appropriate. Also, a combination with possible extensions considered in (1) seems difficult.

(7) Atomicity of transition action

Atomicity principle is strongly related to almost any other topic considered here (see, e.g., (2), (3), (5) and (9)). It seems very complex to correlate them all. In a way, the atomicity of transitions makes other changes easier. On the other hand, as simple constraint as "one output per transition" allows for (almost) unrestricted interleaving between transition parts without changing (in a given sense) the current, overall semantics. It would be nice to have some more realistic and sufficient conditions relaxing the atomicity principle (the sources of difficulties are known).

(8) Synchronisation and similar mechanisms (e.g., blocking send, rendez-vous)

Again, introducing a new mechanism of that sort must be carefully discussed with any changes proposed in (3), (4) or (7). A rendez-vous synchronisation was successfully defined (Estelle*) [1], for a subset of Estelle, and used in some tools like ESTIM simulator [2] or as an input language to Xesar verification [7].

(9) Estelle constructs in Pascal functions and procedures

There is a strong temptation to allow such constructs. It should be realised, however, that this is difficult (or impossible) for functions unless they are not considered to be "pure". In general, the postulate contradicts a silent assumption that Pascal is a support language that could be easily changed for another. Whether the assumption should be kept it is disputable and a matter of taste. With the whole complexity of Pascal it may be difficult to imagine all possible consequences if the inclusion were allowed. The question is worth to be investigated.

(10) Semantics of open specifications

Estelle semantics, for the purpose of the standard, was defined in such a way that environment component was not taken into account (it was not necessary for the explanation of Estelle mechanisms and that was the purpose of the standard). Very easily, the environment can (and clearly should) be included in the semantic definition (some detailed problems are to be discussed). The environment is a (separate from the so called "global instantaneous description") valuation of input queues (top of queues) that are not connected at the moment but have potential input interactions defined. Everything is then defined as before but taking into account these potential valuations. These semantics extensions clearly does not solve practical problems of how to deal with an open module in a real-life context (interfacing). But we feel that the language definition should be separated from other important issues concerning the use of the language. An approach in this practical direction is discussed in [9].

(11) Alternative semantics for the delay-clause and timed transitions

Alternative semantics for delays (i.e. semantics with absolute priority for delayed transitions with upper bound delay elapsed, or semantics that discards these transitions from competition if not chosen for execution in the right time interval) were considered during the language design. Finally the current option was chosen. It must be said that the above options are equally easy (or difficult: nondeterminism and priorities) to define. Therefore, the choice was somehow arbitrary (any choice has its advantages and flaws). In general, the semantics of timed Estelle should be - in our opinion - changed. We propose explicit definition, w.r.t. to given time constraints, in the style defined in [5]. It avoids such strange concepts as "optionally fireable" transitions that led to some unexpected situations (not very probable in practice, but confusing at the definition level) in presence of priorities.

(12) Estelle module destruction statements (detach, release)

Some complains have been heard about complexity of the two above statements. There have been postulates to restrict the possibility to simple-detach and termination (adopted by ISO in last minute). However, any discussion has to be preceded by a decision concerning (4).

II. Concluding remarks

As we mentioned, the above list is incomplete. There are many ways Estelle can be enhanced, corrected or changed (e.g. as proposed in [8]). We feel, however, that the above subjects have been raised most often.

References

[1] J-P. Courtiat, Estelle*, a powerful dialect of Estelle for OSI protocol description, Proc. PSTV'88, North-Holland, 1989

[2] J-P. Courtiat, P. de Saqui-Sannes, ESTIM, Comp. Networks and ISDN Systems 25, 1992

[3] S.C. Chamberlain, P.D. Amer, Broadcast channels in Estelle, IEEE Trans. on Computers 40, 1991

[4] P. Dembinski, Estelle Semantics: Propositions of Changes and Improvements, SEDOS 0.77 Report, 1987

[5] P. Dembinski, Semantics of Timed, Concurrent Systems, Fundamenta Informaticea, vol.29, no. 1,2, 1997

[6] C. Jard, J.F. Monin, R. Groz, Development of Veda: a prototyping tool for distributed algorithms, IEEE, Trans. on Software Engineering 14, 1988

[7] J.L. Richier, C. Rodriguez, J. Sifakis, J. Voiron, Xesar: A Tool for Protocol Validation, User's Guide, Version 1.2, IMAG Reports, 1987

[8] R. L. Tenney, Adding Interaction Sets to Estelle, Proc. FORTE'91, North-Holand, 1992

[9] J. Thees, R. Gotzhein, Open Estelle - An FDT for Open Distributed Systems, Proc. FORTE/PSTV'98, Chapman and Hall, 1998

[10] P. Dembinski, Asynchronous Inter-module Communication, Internal Report, 1995

[11] J. Bredereke, R. Gotzhein, Increasing the concurency in Estelle, Proc. FORTE'93, North-Holand, 1994