Correspondence

Some Comments on "Transition-Oriented" Versus "Structured" Specification of Distributed Algorithms and Protocols

G. v. BOCHMANN AND J. P. VERJUS

Abstract—**Formal description techniques (FDT's) are being developed for the specification of communication protocols and other distributed systems. Some of them (namely SDL and Estelle) are based on an extended state transition model and promote a "transition-oriented" specification style. Another one (namely Lotos) and most high-level programming languages promote a style which is called "structured." The correspondence compares these two specification styles in the framework of rendezvous interactions between different system modules. The advantages of each of the two styles are discussed in relation with an example of a virtual ring mutual exclusion protocol. Transformation rules between the two approaches are given. An extension to the state transition oriented FDT's is also suggested in order to allow for a structured specification style.**

Index Terms—**Distributed algorithms, Estelle, exception handling, extended state machines, mutual exclusion, SDL, structured programming.**

I. Introduction

Formal description techniques (FDT) for the specification of communication protocols and services are being developed by ISO and CCITT to be used in the area of Open System Interworking [14], [3]. It is expected that these techniques could also be used as specification language in other areas of application. One of the FDT's, called Estelle [5], [13], uses a descriptive model based on Pascal and the concept of finite state machines; a similar model is also used by the CCITT specification language SDL [15] (for references to related work, see for instance [1]). Using this FDT, a system is described as consisting of a certain number of "modules," each specified as an extended state machine. The system structure is defined by a (usually static) interconnection pattern, and two interconnected modules may interact through the exchange of "signals" (also called "interactions") which may include parameters. Originally, two options were foreseen in Estelle for the interactions between two given modules: 1) rendezvous interaction, where the "sending" module must wait until the "receiving" module is ready for the reception of the signal, and 2) interaction with queueing, where the signals generated by the sending module are put into a (conceptually) infinite queue and are received by the "receiving" module in FIFO order as soon as it is ready. Only this latter option is now provided in Estelle and SDL. Another FDT, called Lotos [8], is based on CCS [9] and provides only rendezvous as interaction primitive of the language.

At the same time, much research in the area of distributed system specification methods is aimed at a better understanding of the basic problems of distributed system design through the study of such language concepts as CSP [7] and CCS [9]. Other work concerns the systematic derivation of distributed algorithms from the specification of requirements which is often given in a centralized view (see for example [12] and [16]). In this kind of work, a distributed algorithm or protocol is often given in a style, usually called "structured," which corresponds to structured programming practice and its familiar nested program structure, as supported by most modern programming languages.

The purpose of this correspondence is to compare two specification styles which we call "transition-oriented" and "structured," respectively. The first is promoted by Estelle and other state-machine oriented specification methods, where a specification consists essentially of a list of possible state transitions. The second style is related to "structured programming" with appropriate facilities for specifying parallelism, as exemplified by languages such as Ada®, CCS, or Lotos. For this comparison, we assume the original rendezvous option of Estelle, which provides an interprocess communication semantics similar to those of Ada, CCS, and Lotos, based on the concepts of "rendezvous" and "guarded commands." It is shown that a simple extension to Estelle could be defined in such a way as to allow "structured" specifications in Estelle, and such that the "transition-oriented" specification style would appear as a special case of the more general "structured" style.

The correspondence is organized as follows. Section II presents a simple example system and the specification of one of the system modules in the "structured" specification style using (slightly) extended Estelle. A definition of the semantics of this version of Estelle is then given in Section III. The relation between the "structured" and the "transition-oriented" specification styles is explored in Section IV, where transformation rules between the two approaches are discussed. This section also includes a discussion of advantages and disadvantages of these two approaches. Some concluding remarks are given in Section V.

II. An Example

The example considered here is an algorithm which attributes a privilege (for example, access to a resource) in mutual exclusion to a number of **user** modules which communicate with one another in the form of a virtual ring which is supported by a physical network to which the modules are connected. The description given here is based on an original algorithm of Dijkstra [4] further discussed and modified in [10]. A description of the algorithm using Estelle was also given in [6].

The overall system structure is shown in Fig. 1. Each **user** module is connected to the **virtual ring** through an **ME_controller** (mutual exclusion controller) which determines when the user in question may obtain the privilege. The implementation of the **virtual ring** is not considered in this paper; details about the maintenance of the virtual ring structure in the presence of faults may be found in [10] and [6].

The idea of the mutual-exclusion algorithm is that each **ME_controller** module can consult the "state" of its left neighbor on the ring and is able to determine whether it may give the privilege to the user from the knowledge of its own and the neighbor's state. The own state is updated after the privilege has been used.

The possible interactions of the **ME_controller** module are

Manuscript received January 31, 1984; revised June 28, 1985.

G. v. Bochmann is with the Department d'IRO, University of Montreal, C.P. 6128, Montreal, P.Q. H3C 3J7, Canada.

J. P. Verjus is with IRISA, University of Rennes, Rennes, France.

IEEE Log Number 8613171.

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

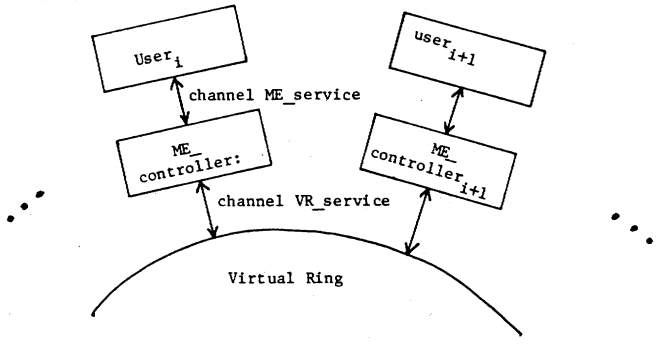

Fig. 1.

specified below using Estelle syntax. For each of the two channels through which the **ME_controller** module may interact, the types of possible interactions are listed with an indication which module may initiate the interaction and select the values of the parameters. For the **ME_service** channel, for example, the two roles **ME_user** and **ME_provider** are specified. The module playing the **ME_user** role initiates the four interactions mentioned in the channel specification, while the other module plays a passive role, receiving these interactions.

```
channel ME_service (ME_user, ME_provider);
   by ME_user: ME_begin; ME_end; F_begin; F_end;
```

In the case of the **VR_service** channel below, both modules have some active part to play. Only the two interactions **S_resp** and **S_conf** have a parameter, which is used to convey state information between adjacent modules on the ring.

```
channel VR_service (ring, user);
   by user: F_begin; F_end; S_req; S_resp (S : state_type);
   by ring: S_ind; S_conf (S : state_type);
```

It is noted that the user initiates the **ME_begin** interaction when it wishes to obtain the privilege. When the **ME_controller** executes this interaction in rendezvous with the user the "privilege" is passed to the user. When the user does not require the privilege any more, it initiates the **ME_end** interaction. In the case of a failure, the user initiates the **F_begin** interaction. The termination of a failure situation is indicated by the **F_end** interaction. Similarly, the **ME_controller** module may indicate failures to the virtual ring. The order of interactions for the exchange of state information between an **ME_controller** module and its "left" neighbor is indicated by the time-sequence diagram in Fig. 2: An **S_req** interaction initiated by the module in question is followed by an **S_ind** initiated by the virtual ring to its neighbor; the state information is returned through the **S_resp** and **S_conf** primitives.

The mutual-exclusion algorithm can be described by the following program which defines the behavior of a **ME_controller** module.

```
module ME_controler; ip up : ME_service (ME_provider);
                        down : VR_service (user) end;
body ME for ME_controler;
   var my_S : state_type;
   begin output down.S_req; while true do begin
         normal : : select
            when down.S_conf (left_S)
               begin
                  if some_predicate(my_S, left_S)
                  then begin
                     privileged : : select
                        when up.ME_begin
                           ME : : begin select
                                    when up.ME_end begin end;
                                    end select
                              end;
                           otherwise begin end;
                     end select;
                     change_state (my_S);
                     end
                  else;
                  output down.S_req
               end;
            when down.S_ind
               begin output down.S_resp(my_S) end;
            when up.F_begin
               begin
                  output down.F_begin;
                  F : : select when up.F_end
                     begin output down.F_end;
                           output down.S_req end;
                  end select;
               end;
      end select;
   end;
```

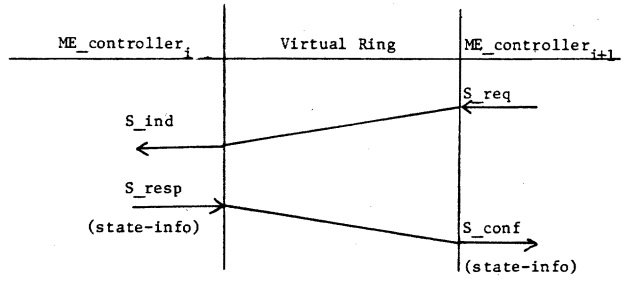

Fig. 2.

The interpretation of the algorithm given above is relatively straightforward. The module sends periodically an **S_ind** over the ring to its left neighbor and waits for one of the following events to happen.

1) The answer from the left neighbor in the form of an **S_conf** interaction: If **some_predicate** is true, the privilege can be given to the user module, since this predicate can only be true for at most one site. The predicate depends on the state of the module itself (variable **my_S**) and the state of the left neighbor which is passed as parameter **left_S** through the **S_conf** interaction. (As explained in [10], the state of a site is composed of two parts: the site number, and a counter variable. The latter is updated by the operation **change_state** which has the effect that the predicate becomes true for the neighbor to the right. In this example the circulation of the privilege is assured by the periodic status requests to the left neighbors; other approaches to ensuring this circulation are described in [6].)

If the predicate is true the privilege is passed to the user only if the latter requested it. Whether the privilege was requested is checked by determining whether the **ME_begin** interaction with the user can be executed or not. If yes, the privileged region (mutual exclusion) is entered until its end is indicated by the execution of the **ME_end** interaction. Otherwise the privilege is immediately passed on to the next site through the execution of the **change_state** operation.

If the predicate is not true there is nothing to do, except a periodic retry of the status request to the left neighbor.

2) Reception of a status request from the right neighbor: A response is returned immediately. Note that the structure of the program implies that such a request cannot be received when the site is in the privileged or failure state.

3) Entrance into a failure state, indicated by the user through the **F_begin** interaction: The module waits that the failure is terminated, as indicated by the reception of a **F_end** interaction from the user over the channel identified by the **up** port. The **F_begin**

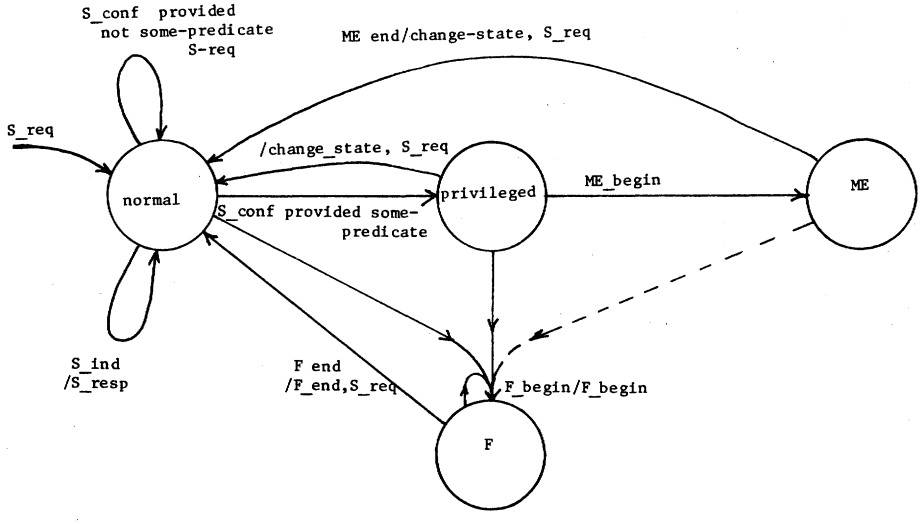

Fig. 3.

and **F_end** indications are propagated to the virtual ring through the **down** port. Note that it is assumed that a failure never occurs directly within the privileged state (see discussion in Section IV).

III. Language Definition

This section provides an informal definition of the language used for describing the example above. The definition is given in comparison to Ada and Estelle (queueing option).

A. The Nature of Module Interaction

Module interactions have the rendezvous semantics as defined for Ada or CSP. This implies that an interaction between two modules can only take place when one module is ready for the execution of an output statement, and the other module is ready for executing an input reception of an interaction of corresponding type and over the same channel. If the latter module is not ready (i.e., executing internal operations, outputting, or waiting for another type of input or over another channel) then the former module has to wait. This may well lead to a deadlock if the system specification allows such a situation to happen.

In contrast to Ada, each interaction allows the transfer of parameters only in *one* direction, from the outputting module to the receiving one. This is also the case for the queuing option of Estelle.

B. Language Definition

The syntax for the definition of the **channels** and the heading of the **module** is taken from Estelle [5], as well as the syntax for the input and output. The only extensions to Estelle is the **select** statement which is described below.

A **select** statement is introduced (replacing the **trans** construct of Estelle). The semantic of this statement is as in Ada, namely the selection of one of the possible "choices" given inside the statement. Two kinds of choices may be included in a **select** statement: 1) input choices which correspond to the reception of an interaction of a specific type over a particular channel, or 2) spontaneous choices which may be chosen based solely on the internal state of the module. (No example of a spontaneous choice is given in this paper.)

A spontaneous choice begins with **provided** <boolean expression> where the expression indicates the condition which must be true for this choice to be selected. If it is selected the following "begin. . .end" statements are executed.

An input choice begins with a **when** clause which indicates the kind of interaction to be received for this choice to be selected. It may also be followed by a **provided** clause which indicates an additional condition depending on the internal module state and the parameters of the received input interaction. The following "begin. . .end" statements indicate the actions to be taken if the choice in question is selected.

If several choices are possible in a given system state, an implementation of the specification will select one of the possible choices; which one is not specified.

The **otherwise** choice indicates that a "begin. . .end" construct is to be executed if on the entry to the select statement none of the explicitely defined choices is possible. This can be considered a special case of static "priorities" associated with input interactions, as defined in Estelle.

Some labels are included in the example of Section II using the notation " label_id :: ". They are only introduced in order to show the relation of the "structured" specification of Section II with the "transition-oriented" specification given in Section IV and shown in Fig. 3.

C. Considering Estelle as a Special Case

Estelle (rendezvous option) may be considered a special case of the language described above. The transitions defined in an Estelle specification may be considered to correspond, one by one, to choices in a single select statement which is repeated indefinitely, according to the following program structure. Such a program structure is called "transition-oriented" in this correspondence.

```
body . . . for . . .;
    var . . .;
    begin while true do select
        <transition 1>;
        <transition 2>;
        . . .
        <last transition>;
    end;
```

IV. Transformation Between "Structured" and "Transition-Oriented" Specifications

The relation between "structured" and transition-oriented" specifications are considered in this section, as well as transformations that go from one specification style to the other. The transformation from a "structured" specification to an equivalent "transition-oriented" one is discussed below. On the other hand, Section III-C takes the view that a "transition-oriented" specification may be considered as a special case of a "structured" one. However, this view ignores the possibility that an equivalent specification may be found with additional structure. Methods for finding such structure are outside the scope of this correspondence.

As far as the transformation of a "structured" specification into a "transition-oriented" one is concerned, it is important to note

that several methods for automatic transformation may be envisaged. Depending on the complexity of the control structures in the programming language used for the "structured" specification, such transformation will become more or less complex.

If the "structured" specification uses no GOTO statements, a transformation methods may be developed using the following approach: A control state variable is introduced, sometimes called "major state" and usually represented by the variable name **state**. The possible values of this variable correspond to places in the program text of the "structured" specification. (In the example, they are indicated by the labels.) The beginning of each **select** statement, in particular, corresponds to a value of this variable. There is at least a transition for each choice of a **select** statement. The action of a transition will usually extend up to the beginning of the next **select** statement in the program text. If a loop of the "structured" specification contains a **select** statement then the loop will be "cut" leading to one or several transitions to be executed for each iteration of the loop. This approach has been used for the example given below.

A. The Example

Using the transformation approach described above, the mutual exclusion algorithm described in a "structured" style in Section II may be rewritten in a "transition-oriented" style as follows. The notation **from** <present major state> **to** <next major state> [5] used below indicates that a given choice (i.e., "transition") is only possible if the present state has a particular value, and it indicates the value of the **state** variable (also called "major state") after the execution of the "begin. . .end" statements of the choice. An overview of the major states and "transitions" of the specification is given in the diagram of Fig. 3.

```
body ME for ME_controller;
   var my_S : state_type;
         STATE normal, privileged, ME, F;

   initialize to normal begin output down.S_req end;

   begin while true do select

      when down.S_conf (left_S)
      provided some_predicate (my_S, left_S)
         from normal to privileged
         begin end;
      when down.S_conf (left_S)
      provided not some_predicate (my_S,left_S)
         from normal to normal
         begin output down.S_req end;
      when up.ME_begin
         from privileged to ME
         begin end;
      provided true priority lower
         (* normally if no ME_begin is waiting *)
         from privileged to normal
         begin change_state (my_S); output down.S_req end;
      when up.ME_end
         from ME to normal
         begin change_state (my_S); output down.S_req end;
      when down.S_ind
         from normal to normal
         begin output down.S_resp (my_S) end;
      when up.F_begin
         from normal, privileged, ME, F to F
         begin output down.F_begin end;
      when up.F_end
         from F to normal
         begin output down.F_end;
               output down.S_req end;
   end select; end;
```

The transition indicated in Fig. 3 by the dashed line is not included in the "structured" specification given in Section II. In fact, for the specification of Section II, it is assumed that a user module never initiates a fault indication when it is in the mutual exclusion state. This assumption is not necessary for the "transition-oriented" specification given above. It is interesting to note that the introduction of the dashed transition poses no "structural problem" in the specification. It is simply "another transition," in this case actually included in the before last transition as one of the cases for the present major state.

B. Exception Handling

The introduction of the equivalent of the dashed transition in the "structured" specification of Section II leads to some statements concerning the failure interactions in the innermost select statement. This results in a somehow "unstructured" specification since considerations of failure would be distributed to two places in the program.

An alternative method for handling this situation is the introduction of an additional control structure to the language for handling "exceptions" with higher priority. Associating an **exception** clause with a statement in the language, and assuming that the scope of the exceptional, high priority choice specified in the clause applies to the whole statement the specification of Section II may be rewritten in the following form.

```
body ME for ME_controler;
   var my_S : state_type;
   initialize begin output down.S_req end;
   begin while true do
      begin select
            when down.S_conf
               . . . see Section 2
            when down.S_ind
               . . . see Section 2
         end select;
      end exception select when up.F_begin
                  begin
                     output down.F_begin;
                     select when up.F_end
                        begin output down.F_end end;
                     end select;
                  end;
         end select;
   end;
```

V. Conclusions

This correspondence compares the "transition-oriented" specification style promoted by certain specification techniques, such as Estelle [5] and SDL [15] and a "structured" specification style based on programming languages, such as Ada or CSP, using rendezvous primitives for interprocess communication. The similarity of the transition-oriented style, when combined with rendezvous for interprocess communication, with the "structured" specification style is pointed out in Section III. The following remarks conclude the discussions of this correspondence.

1) Rendezvous Communication: The rendezvous interaction primitives have the property that the receiving module may determine if and when a particular interaction may be executed. This power is essential for many examples. It allows the writing of "structured" specifications, but care must be taken to avoid the possibility of deadlocks. (This power is not provided by Estelle and SDL, as presently defined.)

2) Nondeterminism: For a specification language, the possibility of leaving certain properties of the specified system undetermined seems important. In the case of the specification language considered here, nondeterminism can be introduced by the undetermined selection of a choice within a **select** statement. However, the nondeterminism is partly reduced by the environment which

may determine the next interaction. Sometimes the nondeterminism is further reduced by defining priorities among the different choices, for instance through the **otherwise** clause used in the example of Section II.

It is important to note that the discussion in this paper does not address the problem of "liveness" (see for example [11]). In the case that a specification allows a choice between several different alternatives, how does one specify that the choice between the alternatives should be fair, that is, each of the choices will eventually be executed, if this is possible at all? It seems that considerations of liveness, as well as performance are usually part of a specification and should be addressed by a specification language.

3) Parallelism: Certain languages allow for the expression of processes or sequences of statements which are executed in parallel, and may share some common data. Not all "structured" languages allow for this possibility. However, in the "transition-oriented" style of specification, such a situation may be expressed by decomposing each of the parallel processes into a number of transitions, such that these transitions belonging to different processes may be executed in an interleaved manner. Although this is not true parallelism, this approach allows nevertheless an arbitrary fine interleaving of the processes depending on the size of the individual transitions.

4) Exceptions: The "transition-oriented" style invites the designer to write a transition of the form

```
provided "some exceptions" from any_state to failed
begin "do exception processing" end;
```

which will be executed in any circumstances when the specified exception occurs. This approach is straightforward and easy to use, however, it covers the fact that for certain systems specific exception processing is required depending on the context in which the exception occurs. In the example above, the occurrence of a failure during the holding of a privilege may require a different processing than in other circumstances. The "structured" specification style forces the designer to consider the different circumstances more explicitly, as discussed in Section IV-C.

References

[1] G. v. Bochmann, E. Cerny, M. Gagne, C. Jard, A. Leveille, C. Lacaille, M. Maksud, K. S. Raghunathan, and B. Sarikaya, "Experience with formal specifications using and extended state transition model," *IEEE Trans. Commun.*, vol. COM-30, pp. 2506–2513, Dec. 1982.

[2] G. v. Bochmann, G. Gerber, and J. M. Serre, "Semi-automatic implementation of communication protocols," Informatique et Recherche Operationnelle, Univ. Montreal, Publ. 518, Dec. 1984.

[3] G. J. Dickson and P. de Chazal, "Application of the CCITT SDL to protocol specification," *Proc. IEEE*, vol. 71, Dec. 1983.

[4] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, Nov. 1974.

[5] "Estelle: A FDT based on an extended state transition model," ISO TC97/SC21, DP9074, 1986.

[6] R. Groz, "Description de l'algorithme de Mossiere-Tchwente-Verjus avec le langage de description FDT de l'ISO et du CCITT," CNET/EVP, Lannion, France, Note Interne, 1983.

[7] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[8] "Lotos: A formal description technique," ISO TC97/SC21, DP8807, 1986.

[9] R. Milner, *A Calculus of Communicating Systems* (*Lecture Notes in Computer Science*, No. 92). Berlin: Springer-Verlag, 1980.

[10] J. Mossiere, J. Tchente, and J. P. Verjus, "Sur l'exclusion mutuelle dans les reseaux informatiques," IRISA, Rennes, France, Publ. 75, 1977.

[11] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 455–495, July 1982.

[12] Y. Paker and J. P. Verjus, Eds., *Distributed Computing Systems, Synchronization, Control and Communication*. New York: Academic, 1983.

[13] R. Tenney, "One formal description technique for OSI," in *Proc. ICC*, 1983.

[14] C. A. Vissers, G. v. Bochmann, and R. L. Tenney, "Formal description techniques by ISO/TC97/SC16/WG1 ad hoc group on FDT," *Proc. IEEE*, vol. 71, pp. 1356–1364, Dec. 1983.

[15] *CCITT Recommendation Z. 100*, CCITT SDL, 1987.

[16] G. v. Bochmann and R. Gotshein, "Deriving protocol specifications from service specifications," in *Proc. Symp. Commun. Architectures and Protocols* (*ACM*), Aug. 1986.