DELTA
Project Report No. 6

Implementation of the Delta Language Interrupt Concept within the Quasiparallel Environment of Simula

by
Morten Kyng

DAIMI PB-58
August 1976

Institute of Mathematics University of Aarhus
DEPARTMENT OF COMPUTER SCIENCE
Ny Munkegade - 8000 Aarhus C - Denmark
Phone 06-12 83 55

CONTENTS

1. SYSTEM DESCRIPTION AND PROGRAMMING

The DELTA Language is a language for the description of systems and for communication about systems (1).

When a person wants to communicate information about some referent system he may produce a DELTA system description of the system. The description contains the aspects of the referent system, which he regards as essential for the understanding of the system (with respect to the purpose of making the description).

A person receiving the DELTA system description may then use it to generate and operate a model system, possibly in his mind, i.e. on the substrate of his brain, or on a blackboard, a piece of paper etc.

The person generating the model acts as a "generalized computer", a DELTA System Generator, executing a DELTA system description in much the same way as a computer executes a SIMULA program.

The DELTA Language is however not a programming language. Indeed it was anticipated from the beginning of the DELTA Project, that several of the major aims could be reached only by abolishing the restrictions necessarily placed on any programming language for a digital computer.

The systems described in the language contain components, objects, executing actions in parallel. The actions are instantaneous or time consuming and the time consuming actions may constitute continuous state transformations, i.e. they may result in the continuous change of the value of one or more variables.

Though the DELTA Language is not a programming language, SIMULA 67 was an important part of the starting platform of the project,

(1) The DELTA Language is described in Holbæk-Hanssen, Håndlykken and Nygaard (1975). The subset of the language presented here is somewhat simplified. However, all aspects relevant to our discussion of projection into SIMULA's quasiparallel environment are treated without simplifications.

and the system description aspect forms a major part of several of the SIMULA applications (Nygaard, 1973).

Because of this it seems worthwhile to investigate the projection of the DELTA Language concepts into the quasiparallel environment of SIMULA; and thereby benefit from the experience gained in a truly parallel environment, much more like the environment of the systems we want to describe in SIMULA, than the environment of SIMULA itself.

<u>2. THE DELTA CONCEPTS</u>

<u>Nest Structured Systems</u>

The systems described in the DELTA Language are nest structured systems. The system as a whole is represented by an object, the SYSTEM object. The system object then contains the other objects of the system. But it need not be the direct encloser of all the other objects. Consider a SYSTEM object containing a collection of PERSON objects. We want the objects representing the heart, lungs, kidneys etc. of a person to be contained in the corresponding PERSON object and not directly in the SYSTEM object.

We regard the PERSON objects as representing subsystems within the original system.

At a later stage of decomposition we may want to supply some organs with an inner structure represented by objects contained directly within the corresponding ORGAN object, and not directly within the PERSON (fig. 2. 1). This process of decomposing an object may be repeated any number of times. In general any object in a DELTA system may contain other objects. We say that an object is the <u>encloser</u> of the objects which it directly contains.

<u>The Action Sequence of an Object</u>

When an object is generated in a DELTA system it immediately begins executing the actions of the <u>prime task</u> described in its object descriptor. Speaking in terms of SIMULA,an object becomes detached and active as soon as it is generated. We say that the object is <u>operating</u>. When all the actions of the prime task are executed the object is <u>terminated</u>.

When an object terminates this should effectively terminate the subsystem represented by the object. Consequently, if the object contains any operating objects these are also terminated.

An object in a DELTA system executes one sequence of actions. When one action is completed the object continues with the next action

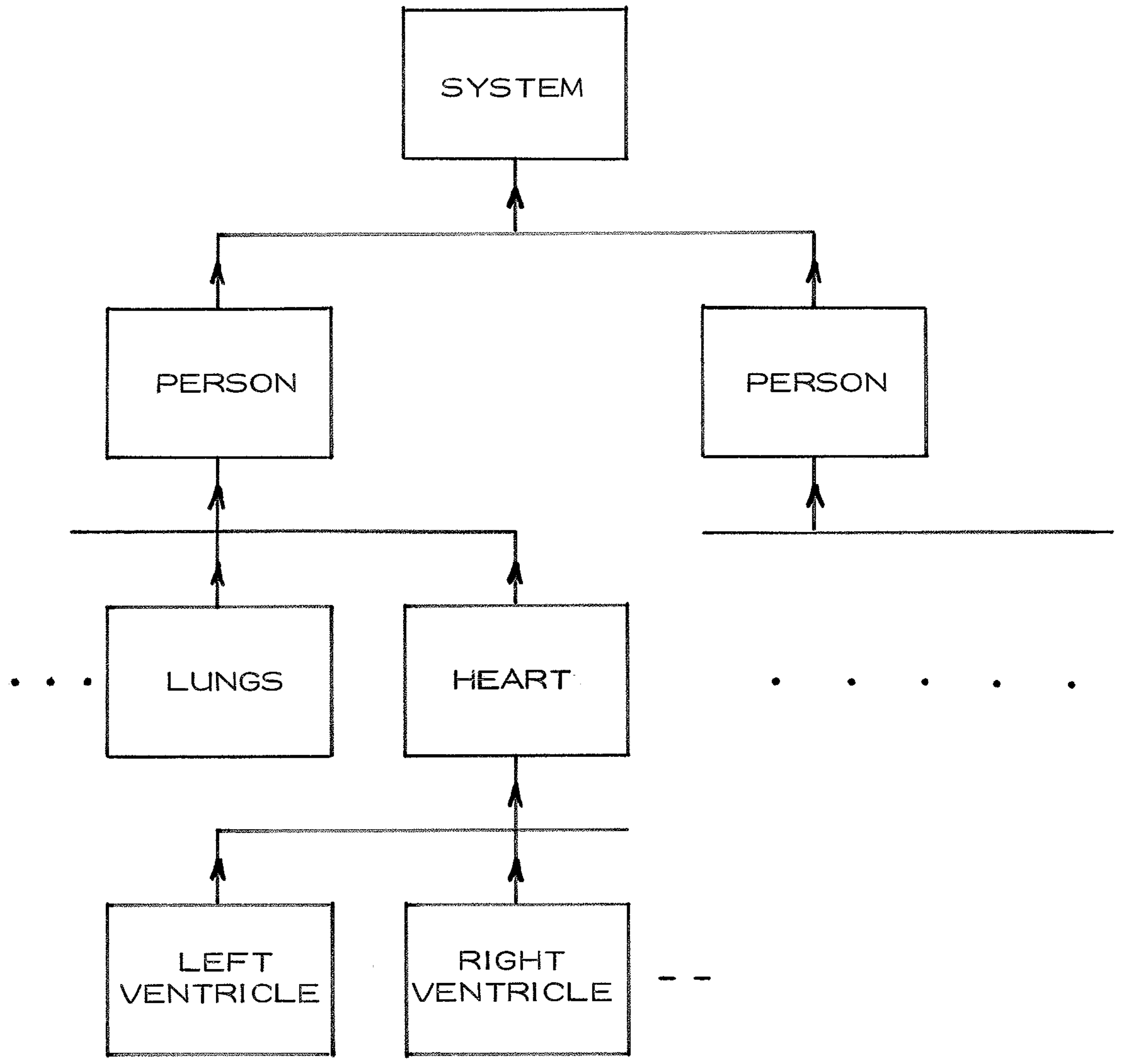

Fig. 2. 1
The structure of a DELTA system

The (direct) encloser of an object is indicated by a directed line in its sequence. If the action sequence contains related actions these may be grouped together to form a _partial task_.

The execution of a sequence of closely related actions (as e.g. the prime task or one of its partial tasks) is represented by an _action entity_. Action entities are drawn as rectangles containing text and connected by directed lines. (cf. fig. 2.2 p. 6).

We may illustrate this situation by a PERSON object with the prime task of eating breakfast. As partial tasks of eating breakfast he may eat a boiled egg and drink a glass of milk. The partial task of eating the egg may in turn include the action of cutting the top off the egg and the action of sprinkling salt on it.

The task of eating the egg and the other partial tasks of the meal are logically related to each other and form together what we name an _activity_.

Each action executed by an object as a part of an activity is either _time consuming_ or _instantaneous_. When an object is executing a time consuming action it may be _interrupted_. The object then temporarily postpones the activity in which it was engaged and starts executing the actions enforced upon it by the interrupt.

This situation is illustrated by the PERSON executing the time consuming action of sprinkling salt on his egg, when interrupted by the door bell. He then postpones the meal and starts executing the actions necessary to answer the bell. He thereby creates a new activity which has no logical connection to the activity of eating breakfast.

When the new activity, that is the actions of the interrupt, is completed, the PERSON returns to the interrupted time consuming action of sprinkling salt on his egg, which resumes the activity of eating breakfast.

We therefore describe the activities of an object as a _stack of activities_, each activity being a stack itself (possibly with only one element). The object as a whole in relation to its activities is represented by an object head (fig. 2. 2):

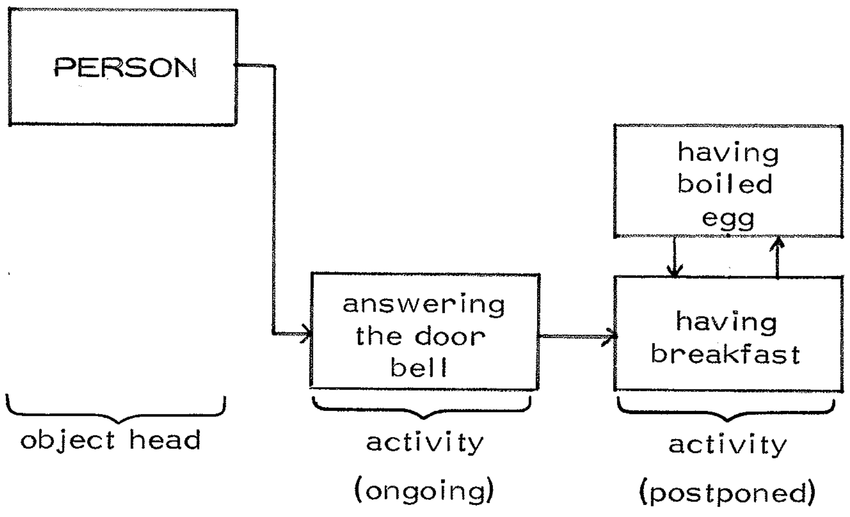

Fig. 2. 2
The activities of an object.

Let us now assume that the PERSON, while sprinkling salt on his egg, has been interrupted by a phone call. The activity of eating breakfast is postponed and the ongoing activity is that of answering the phone. If he now, while talking on the phone, receives an interrupt from the door bell, he may consider the ongoing action so important that he will not answer the bell.

In DELTA this situation is handled by assigning <u>resistance priorities</u> to time consuming actions and <u>power priorities</u> to interrupts. Whether an interrupt <u>penetrates</u> or not, depends upon a comparison of the power of the interrupt and the resistance of the ongoing time consuming action.

An interrupt that does not penetrate immediately is placed on the <u>agenda</u> of the object receiving the interrupt. Later, when the object starts executing another time consuming action with a possibly different resistance, the interrupt may penetrate. It is then removed from the agenda, and the object starts executing the actions of the interrupt, thereby creating a new activity.

A DELTA object thus consists of

- an object head representing the permanent characteristics of the object
- a stack of activities representing the actions in which the object is engaged, each activity being a stack itself, and
- an agenda of received but not yet executed interrupts, organized as a one way list.

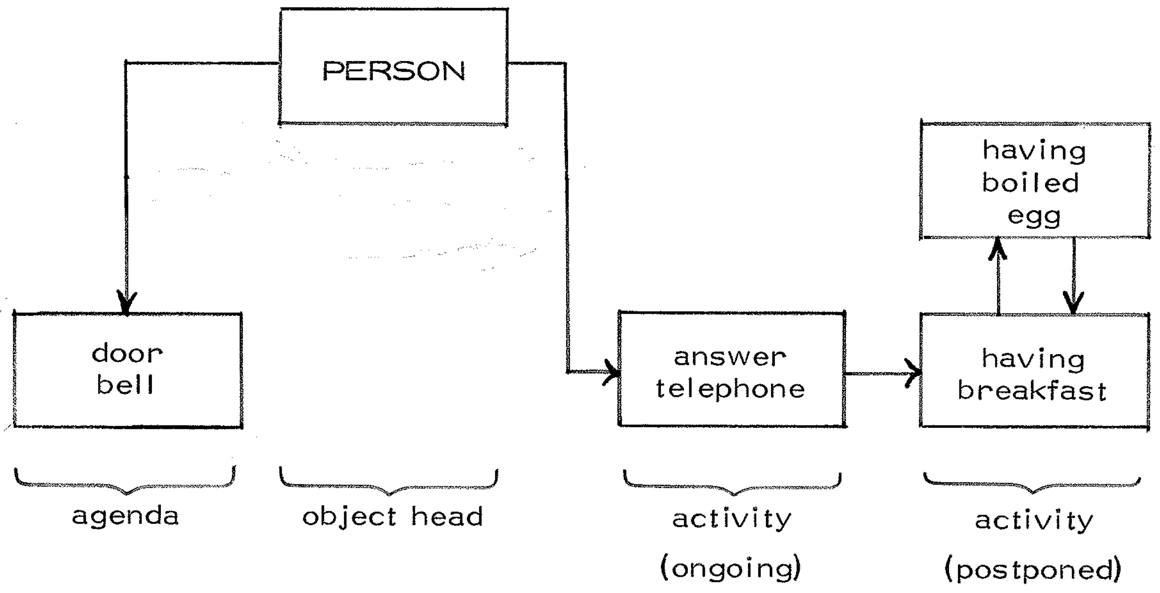

Fig. 2.3
The DELTA object structure

The Actions of an Object and their Description

The actions executed by an object are either time consuming or instantaneous. The time consuming actions are described by time concurrency imperatives as e.g.

$$\underline{\text{WHILE}}\ \text{TIME} < \text{T}\ \underline{\text{LET}}\ \{X^2 \leq F(Y)\}$$

or

WHILE hungry and more food
LET {the meal go on}

A time concurrency imperative consists of up to five clauses:

The duration clause which describes the condition for the continuation of the action. The clause may consist of the keyword WHILE followed by a condition (as in the two examples above). In this case the action continues as long as the condition is fulfilled, unless interrupted. An empty duration clause is equivalent to "WHILE TRUE".

The time property clause describing the property which the action imposes on the system. It consists of the keyword LET followed by a text describing the imposed property and enclosed in braces. The text and the enclosing braces are called a property descriptor. The time consuming action which imposes no property on the system may be described by the time property clause consisting only of the keyword WAIT.

The last three clauses are related to the possible interruption of the action. They are optional.

The resistance clause defines the objects resistance against being interrupted while executing the time concurrency imperative. It consists of the keyword RESISTANCE followed by a priority value. The priority value must be one of a set of values specified in the priority value declaration of the object. This declaration consists of a list of identifiers indicating the priority values, together with a specification of the "penetration relation" between each pair of priority values.

The postponement clause which describes the actions to be executed if the time consuming action is postponed because an interrupt penetrates. The actions of the clause are executed prior to the initiation of the interrupting task. The clause consists of the keyword EXIT followed by a description of the postponement actions.

The resumption clause which describes the actions to be executed when the time consuming action is resumed after an interrupt. The actions are executed after the completion of the actions of the interrupt and prior to the resumption of the time consuming action. The clause consists of the keyword REENTRY followed by a description of the resumption actions.

A special keyword, ADVANCE, may be used within the resumption clause to describe the situations where the interrupted action should not be resumed, but execution should continue with the action following the interrupted action.

The following example contains all five clauses:

```
WHILE not enough LET {sprinkling of salt go on}
RESISTANCE LOW
EXIT (* put the salt shaker on the table *)
REENTRY (* take the salt shaker *)
```

Instantaneous actions may be described by algorithmic imperatives (statements) as those of SIMULA,possibly mixed with informal language, as indicated in the example above, e.g.

```
        IF TIME < T THEN X. INTO (QUEUE)
or
        IF tired THEN leave the queue.
```

Then sending of an interrupt is considered an instantaneous action. It is described by imperatives as e.g.

```
INTERRUPT JOHN BY TELEPHONE CALL
```

where it is assumed that JOHN is a reference to an object, and that TELEFONE CALL is a procedure describing the task, which JOHN should execute when the interrupt penetrates. Since no power priority is explicitly assigned, the interrupt will get the default priority value NO and may only penetrate time consuming actions with resistance priority NO. Other priority values may be assigned by a penetration clause. It consists of the keyword POWER followed by the priority value, which must be one of a set of values specified in the priority value declaration of the object receiving the interrupt.

Property descriptors may also be used in the description of instantaneous actions. A discussion of these is outside the scope of this paper.

The Operation of a DELTA System

In the preceding two subsections we have discussed the actions within a DELTA system from the point of view of an object.

As stated in section 1, a DELTA model system is, however, generated and operated by a DELTA System Generator (DSG) according to a DELTA system description. In the following subsections we will introduce the concepts necessary to describe how the DSG operates.

The Time Concept

If we consider the time spent by a DSG to execute a system description, we refer to the time in the invironment of the DSG, which we name generator time.

A sequence of actions, which in the referent system take days or weeks of referent time, may be portrayed by actions in the model system which are executed in seconds of generator time. To handle this situation we associate a special non-decreasing variable, model time (or just TIME), with the model system. TIME is operated by the DSG to portray referent time.

The concepts of time consuming and instantaneous actions refer to the time in the referent and the model system. It is possible that the DSG uses generator time to execute what we consider as an instantaneous action. It is sufficient that the value of TIME is kept constant.

System States and Events

We now consider the correspondence between actions and between states in the referent and the model system.

By the state of a system we will understand the set of values of all the quantities and references associated with the objects contained in the system plus the stage of execution of the objects, considered at a given moment.

The stage of execution of an object is the position it has reached within its action sequence. If the object at the moment considered is executing a time consuming action, its stage of execution is that ongoing action. If it is not executing a time consuming action we,consider its stage of execution as the position succeeding its last executed action (i. e. we do not consider the execution of an instantaneous action as a well defined stage of execution).

When all the operating objects in a system are executing time consuming actions, we say that the system is in a concurrent state. At moments when one or more of the objects are not executing a time consuming action, we say that the system is in an event internal state (the reason for choosing this name will be discussed below).

A state in the model system which represents or corresponds to a state in the referent system is called a representative state.

In the examples discussed so far the correspondence between actions in the referent system and in the model system has been obvious. With respect to time consuming actions we demand that this shall always be the case. Consequently we require that all concurrent states are also representative.

However, a single instantaneous action in a referent system may have to be represented by a sequence of instantaneous actions in a model system. The states existing between the execution of two actions in a sequence representing a single instantaneous action may not be representative:

Consider a referent system where a person waits in a queue, then leaves the queue and finally walks away. We may want to regard his leaving the queue as an instantaneous action. In a model system, the leaving of the queue may be achieved by a sequence of instantaneous actions which adjusts various "predecessor" and "successor" reference variables. The states existing between two instantaneous actions in this sequence do not correspond to states in the referent system, i. e. they are not representative, and have no significance but to the object leaving the queue. (The total, resulting effect of the sequence of actions may of course be significant to other objects).

To ensure that the sequence of instantaneous actions results in a meaningful representative state, it is necessary that the object is not interrupted during the execution of the sequence. (Consider for instance an interrupt specifying that the object enters some queue). But it is not sufficient that the object is not interrupted. It is necessary that two such sequences of actions are executed one after the other: If the object executes the actions in parallel with another object, which modifies one or more of the same "predecessor" or "successor" variables, the result may be meaningless.

In general we regard an object's execution of a sequence of instantaneous actions as a change of one concurrent state to another. In most cases the object is executing a time consuming action in the first of the concurrent states. The change begins by ending this time consuming action (i. e. the property descriptor of the action is no longer effective), then the "frozen" concurrent state is modified by the sequence of instantaneous actions and the change ends by initiating the time consuming action following the sequence of instantaneous actions. The object is now executing a new time consuming action (i. e. the property descripter of the action is effective) and the system is once more in a concurrent state.

Such a change is called an <u>event</u>, and based on our discussion we state that

- the actions of an event are executed as an uninterruptable sequence,

- events are not executed in parallel,

- the states existing during the execution of an event, the event internal states, need not be representative.

(The first (last) event executed by an object differs slightly from the case discussed above since it does not begin (end) with the ending (initiation) of a time consuming action.)

The above discussion illustrates an importent difference between the time consuming and the instantaneous actions introduced so far:

Time consuming actions are executed concurrently with other actions in the system. We say that a time consuming action is a concurrency action.

Instantaneous actions are executed as a part of an event and not in parallel with other actions. We say that such an action is an event action.

In some cases an object executes a sequence of instantaneous actions which results in a representative state after which the object continues with another sequence of instantaneous actions. Between such sequences the object may execute a concurrency action, which

- takes no TIME, but
- like a time consuming action
  - is executed concurrently with other actions in the system,
  - causes a concurrent state to obtain,
  - is initiated and ended by two different events.

We will not discuss these actions further. Their semantics may almost entirely be deduced by considering a time consuming action whose associated condition becomes false at the same moment of TIME as the action was initiated.

The Operation of the DELTA System Generator

When a DSG operates a model system it executes the action sequences of a changing collection of objects. We say that the DSG's mode of operation is multisequential.

Within the multisequential mode we distinguish between two modes of operation:

In the open intervals of TIME where all the operating objects are executing time consuming actions, we say that the DSG operates in the concurrent state transition mode.

In this mode the changes in the system are the result of the concurrent execution of all the ongoing time consuming actions. The task of the DSG is to fulfil simultaneously the set of property descriptors of the ongoing actions, which we name the EFFECTIVE DESCRIPTORS, and to supervise their associated conditions, called the SUPERVISED CONDITIONS.

The concurrent state transition mode is ended when one or more of the objects is to execute an event, i.e. when some of the SUPERVISED CONDITIONS become false and their associated time consuming actions are to be ended.

At these moments of TIME, where one or more of the objects in a DELTA system are executing events, we say that the DSG operates in the unisequential mode.

When operating in this mode the DSG executes one sequence of events, since events must not be executed in parallel.

When the DSG enters the unisequential mode, the system is in a "frozen" concurrent state. The DSG starts by examining each operating object in turn to determine whether the system state causes the object to execute an event or not. If an event is to be executed, the DSG registers this by placing an EVENT NOTICE on the EVENT LIST. The EVENT LIST is a list of EVENT NOTICEs representing events scheduled for execution at the moment of TIME considered. Each object will have at most one scheduled event, i.e. if an object already has a notice on the EVENT LIST, the state will not cause the DSG to register another event for that object.

We distinguish between four different concurrent states causing the DSG to register an event for an object:

- if the encloser of the object is terminated, a termination event is registered,

- if the object has not executed any actions yet, an initiation event is registered,

- if the condition of the ongoing time consuming action of the object is false, a completion event is registered,
- if an interrupt on the agenda of the object penetrates its ongoing action, an interruption event is registered.

Since at most one event may be scheduled for each operating object, it is only necessary to examine the operating objects without an EVENT NOTICE on the EVENT LIST when registering events.

For each of these objects the tests to determine whether an event should be registered or not are executed in the above mentioned sequence. One of the implications of this is,that the DSG only registers an interruption event when the condition of the action to be interrupted is true. (Otherwise a completion event would have been registered).

When the DSG has registered all the events caused by the existing concurrent state, an EVENT NOTICE on the EVENT LIST is chosen and the corresponding event executed. We will have no information about how the DSG chooses. All we know is that one event is chosen.

When the DSG executes an event on behalf of an object, this object passes through a sequence of event internal states. These may not be representative and should have no significance but to the considered object. Only when the event is finished a new concurrent state is established and supervised by the DSG to register events.

Once an event has been scheduled we demand that it is also executed and thus the DSG does not "de-register" events. This rule reduces the impact of the sequence chosen by the DSG on the behaviour of the DELTA system it operates.

When there are no more events to be executed at the moment of TIME considered,the DSG will in an open interval of TIME operate in the concurrent state transition mode.

We informally describe the task carried out by the DSG by

```
TASK BEGIN
        generate the initial DELTA system;
        WHILE the system object is operating
        REPEAT (* register events;
                  WHILE EVENT LIST is not empty
                  REPEAT (* choose and execute an
                            event;
                            impose a concurrent state
                            which satisfy the EFFECTIVE
                            DESCRIPTORS;
                            register events *);
                  WHILE all SUPERVISED CONDITIONS
                           are fulfilled
                  LET {EFFECTIVE DESCRIPTORS have effect}
                  and increase TIME continuously
               *)
END TASK
```

Description 2. 1
The task of the DELTA System Generator

Note the difference between the two first and the last "WHILE-clause". The two first are used in repetitive constructs, and tested prior to each execution of the repeated imperatives, which are grouped together by the parentheses "(*" and "*)" . The last "WHILE" is part of the duration clause of a time concurrency imperative and the succeeding condition is supervised continuously during the execution of the action.

Finally it should be mentioned that functions are not allowed to have sideeffects. Consequently no variables are changed as an effect of evaluating a condition.

3. PROJECTION INTO SIMULA's QUASIPARALLEL ENVIRONMENT

In this section we will investigate how to portray the execution of a DELTA system description by the execution of a SIMULA program.

We will do this in such a way that we are able to interpret the resulting execution as done by a Quasiparallel System Generator (QSG) which models the actions of the DSG.

Concurrent State Transition Mode

The main restriction when projecting into SIMULA occurs in the concurrent state transition mode. When operating in this mode the DSG portrays continuous change and supervises the set of conditions associated with the ongoing time consuming actions.

All the states which obtain in an open interval of TIME where the DSG operates in the concurrent state transition mode are concurrent and thus also representative.

When using a digital computer the mode of operation is discrete and it is not possible to portray continuous change. Consequently it is not possible to operate a model system in such a way that representative states obtain over an interval of TIME if continuous change is implied:

Consider a DELTA model system containing a variable whose value is changed continuously by the DSG to portray the position of a car.

On a computer we are not able to execute a time consuming action, which results in the correct change of the value of the considered variable over a period of TIME. We may at most approximate the change in such a way that the value of the variable corresponds to the position of the car at discrete points on the TIME axis and not in intervals.

If we prohibit explicit and implicit reference to TIME in the property descriptors we have in fact excluded continuous change.

This in turn implies that the last concurrent state imposed in the unisequential mode will obtain during the following interval of concurrent state transition mode, except for the increase in the value of TIME. That is, TIME is the only variable whose value changes in the concurrent state transition mode. Consequently, when this mode of operation is ended because one or more of the supervised conditions are no longer fulfilled, this has to be an effect of the increase of TIME.

We will therefore classify the conditions according to their dependence upon TIME (cf. "HOLD" and "PASSIVATE" of SIMULATION):

1. TIME dependent conditions, which we name the TCs. These are the conditions depending upon TIME in such a way that the increase of TIME alone may make them false, as e.g.

   $\text{TIME} < T$

   and $\text{TIME} < T \ \underline{\text{AND}}\ X < F(Y)$.

2. Event dependent conditions, which we name the ECs. An EC requires the execution of at least one event action (i.e. an instantaneous action executed as a part of an event) or the imposition of a concurrent state (which follows each event) to become false [1], as e.g.

   $X < F(Y)$

   and $\text{TIME} < T \ \underline{\text{OR}}\ X < F(Y)$.

In the above examples we have assumed that all the inequalities are valid. Indeed the event action "X:=F(Y)" will change the second EC into a TC, and "X:=F(Y)-1" will change it back into an EC.

(1) In the following subsection we exclude the possibility of changing the value of any variable when imposing a concurrent state in the unisequential mode. Thus an EC will require the execution of at least one event action to become false.

When an interval of concurrent state transition mode ends one or more of the SUPERVISED CONDITIONS are no longer fulfilled. Since TIME is the only variable whose value changes in this mode, the conditions which have become false will all be TCs. This implies that only the TCs need to be supervised in the concurrent state transition mode.

The question still remains of how to supervise the TCs, that is how to increase the value of TIME.

As already stated we cannot increase the value of TIME continuously.

One possibility would be to approximate the continuous increase by adding a small increment to the current value of TIME and then, for each new value of TIME, check all the TCs to see if any one of them had become false. We rule out this possibility, primarily because it does not allow for an exact association of a state in a DELTA system with its projection into an execution of a SIMULA program:

The value of TIME when the Quasiparallel SG ended a period of concurrent state transition would only approximate the value of TIME when the DSG ended its corresponding period.

Instead we place the restriction on the TCs that it shall be possible to compute the value of TIME when the condition becomes false, if no events happen in the system, i. e. if the only change in the system is the increase of TIME.

The main task of the QSG inthe concurrent state transition mode is then reduced to the scanning of the TCs computing for each condition when in the future it will become false. The QSG then sets TIME to the minimum of these values and begins executing in the unisequential mode.

Unisequential mode

In the unisequential mode the DSG executes a sequence of events. After the execution of each of the events it establishes a concurrent state, which satisfies the EFFECTIVE DESCRIPTORS.

Consider an object, OBJ1, executing an event that ends by the initiation of the action

$$\underline{\text{WHILE}}\ \text{TIME} < \text{T}\ \underline{\text{LET}}\ \{X^2 = 1\}$$

When a concurrent state has been imposed the value of X may be -1 . If there are no more events to be exectued at the moment of TIME considered, then the value of X will be -1 in the following interval of concurrent state transition mode.

Now assume that an object, OBJ2, in the following unisequential mode executes an event that ends by the initiation of the action

$$\underline{\text{WHILE}}\ \text{TIME} < \text{T}\ \underline{\text{LET}}\ \{X > 0\}$$

If OBJ1 is still executing the above mentioned action, then the value of X will be 1 when a concurrent state is imposed.

It is possible to conceive a QSG capable of solving and satisfying simple sets of equations and inequalities as those discussed above.

In general, however, it is not feasible for a QSG to impose a concurrent state which satisfies the EFFECTIVE DESCRIPTORS, if the only restriction on the property descriptors is that they must not involve TIME.

In this paper we will not treat the subject of defining such restricted categories of property descriptors and QSGs able to manipulate them.

Instead we state that only the empty property descriptor is allowed. Thus the only time consuming action which the QSG may portray is "no action". This "action" is in the DELTA Language described by the keyword WAIT.

This implies that the action

"impose a concurrent state which satisfies the EFFECTIVE DESCRIPTORS"

executed in the unisequential mode (cf. Description 2.1 p. 16) is empty, i.e. the last action of each event establishes a concurrent state.

Thus every change in the state of the system is an effect of an event action, except the increase of the value of TIME.

We may now modify description 2.1, p. 16 of the DSG to a description of the QSG:

```
TASK BEGIN
    generate the initial DELTA system;
    WHILE the system object is operating
    REPEAT (* register events;
              WHILE EVENT LIST is not empty
              REPEAT (* choose and execute
                        an event;
                        register events *);
              compute the new value of TIME by
              scanning the TCs;
              assign this value to TIME
           *)
END TASK
```

Description 3.1

The task of the Quasiparallel SG

4. ADAPTION OF THE QSG FOR EFFICIENT IMPLEMENTATION

So far we have been concerned with the logical problems of portraying the execution of a DELTA system description by the execution of a SIMULA program.

The description of the QSG given so far is informal, and a straightforward formalization will yield an unacceptable inefficient result, since each event is followed by a scan of all the operating objects without an EVENT NOTICE on the EVENT LIST.

In the following we discuss how to improve the efficiency of the QSG, that is how to reduce the number of objects scanned after each event to register new events.

To enable the QSG to manipulate these objects, we provide it with a TEST LIST. In each concurrent state the list consists of TEST NOTICEs representing objects for which the considered state may cause the registration of an event. During the execution of an event the QSG should place TEST NOTICEs on the TEST LIST for all objects possibly affected by the event. (We will say that the QSG registers tests for these objects.)

Efficiency may be improved further, if the QSG can register events directly during the execution of an event, but this is not feasible since single event actions do not cause events to be registered, only concurrent states do.

Let us, for the time being, assume that the QSG is able to register the necessary tests during the execution of an event, and consider the concurrent state transition mode.

When operating in this mode the QSG uses a list of the time dependent conditions, the TCs, of the actions under execution. We name this list the FUTURE EVENT LIST (the reason for choosing this name will be discussed below). As already described the FUTURE EVENT LIST is used to determine the new value of TIME.

When the value of TIME has been increased a new concurrent state obtains and the QSG should register the events caused by this state before entering the unisequential mode. The objects in question are those with a false condition, i.e. a false TC.

The FUTURE EVENT LIST is built during the unisequential mode. When an object initiates a time consuming action with a TC this includes

- a computation of when in the future the TC will become false. We name this value EVENT TIME. (If EVENT TIME is less than TIME, the TC is initially false and the time consuming action is skipped rather than initiated, and this is why EVENT TIME has to be computed at this moment),
- insertion in the FUTURE EVENT LIST of a notice which refers to the TC.

Since EVENT TIME has to be computed before insertion, we decide to keep the FUTURE EVENT LIST sorted according to EVENT TIME. This implies that EVENT TIME should be an attribute of the items on the list. For this purpose we supply the EVENT NOTICEs with a real valued attribute EVENT TIME. This allows us to build both EVENT LIST and FUTURE EVENT LIST from EVENT NOTICEs which in turn reduces the registration of events in the concurrent state transition mode to a simple transfer of EVENT NOTICEs between the two lists.

We may regard a notice on the FUTURE EVENT LIST as an event scheduled in the future. It may be necessary to change the value of EVENT TIME for some of the notices on FUTURE EVENT LIST and even to remove some, e.g. if a condition is changed from a TC to an EC:

Consider an object executing the action

```
WHILE TIME < T1 OR I < J WAIT.
```

If the first inequality is valid and the last is not, then the object will have a notice on the FUTURE EVENT LIST with EVENT TIME equal to T1. If another object executes the event action

```
I : = J - 1
```

then the above condition changes from a TC to an EC and the notice on the FUTURE EVENT LIST is removed. The condition may now only become false after the execution of another event action which changes the value of I or J.

We may now summarize the preceding discussion in a revised description of the QSG.

The registration of events in the unisequential mode is done by the procedure TEST FOR EVENT. It removes the first notice on the TEST LIST, tests whether the associated object is to execute an event and if so registers the event.

```
TASK BEGIN
          EVENT LIST:  a list of EVENT NOTICEs representing objects
                       for which an event has been scheduled for exe-
                       cution at the moment of TIME considered;
          TEST LIST:   a list of TEST NOTICEs representing objects
                       which may have an event to execute at the mo-
                       ment of TIME considered;
          FUTURE EVENT LIST: a list of EVENT NOTICEs representing
                       objects for which an event has been scheduled
                       in the future;

          PROCEDURE TEST FOR EVENT: ....;
          PROCEDURE EXECUTE EVENT AND REGISTER TESTS: ....;

          generate the initial DELTA system and register the necessary
          initiation events;

          WHILE the system object is operating
          REPEAT
            (* WHILE EVENT LIST is not empty
               REPEAT (* EXECUTE EVENT AND REGISTER TESTS;
                         WHILE TEST LIST is not empty
                         REPEAT (* TEST FOR EVENT *)
                       *);
               IF FUTURE EVENT LIST is not empty
               THEN (* TIME:=
                          FUTURE EVENT LIST. FIRST. EVENT TIME;
                       transfer EVENT NOTICEs with EVENT TIME =
                       TIME from FUTURE EVENT LIST to EVENT LIST *)
               ELSE (* terminate the system *)
            *)
END TASK
```

Description 3.2

The task of Quasiparallel SG

The Registration of Tests

In the preceding discussion leading up to Description 3.2 we have assumed that the QSG were capable of registering tests in such a way that no events was missed. That is, whenever the QSG of Description 3.1 registers an event, then the QSG of Description 3.2 (or just QSG) will do the same.

In the unisequential mode the QSG of Description 3.2 may only register an event for an object if it has a notice on the TEST LIST. Consequently if the QSG of Description 3.1 registers an event for an object in this mode then the QSG should already have registered a test for that object, thereby enabling itself to register the event.

To decide how to register tests during the execution of an event, it is necessary to examine the states causing events to be registered more carefully.

Since an object has at most one event scheduled for execution, it may be possible to arrange the registration of tests and events in such a way that an object has at most one notice on the lists EVENT LIST, TEST LIST and FUTURE EVENT LIST. As we shall see, this is the case.

As stated on page 14 we distinguish between four different states causing the system generator to register an event for an object. We will now consider these in turn.

Encloser termination. When an object terminates a termination event should be registered for each of the enclosed objects in the concurrent state succeeding the event that terminated the enclosing object. This is not possible, unless we associate a list of enclosed objects with each encloser. This may be a rather expensive solution since such lists are used for no other purpose.

Let us consider the consequences of not explicitly registering a termination event when the encloser of an object terminates.

The only way an object may change the state of the system is by an event action and indirectly by allowing enclosed objects to execute events. It is thus sufficient that we prevent the registration of events other than termination events for objects which have an encloser which would have been terminated by the QSG of Description 3. 1 (but possibly is not terminated by the considered QSG of Description 3.2).

If we assume that the encloser is in fact terminated, then the registration of any events other than termination events is effectively prevented by the initial test for encloser termination in each test for registration of an event (cf. p. 15).

Although this assumption does not hold in general, it is valid for the systems which we may portray by the execution of a SIMULA program. The reason for this is, that these systems may only consist of a system object and objects directly enclosed by the system object (cf. p. 34).

We will not go into details with the problems caused by nested systems in general, but only state that they are sufficiently complex, to justify a reevaluation of the solution involving explicit lists of enclosed objects.

Initiation. When an object is generated, this action should be succeeded by the insertion of a TEST NOTICE on the TEST LIST representing the just generated object. In the concurrent state succeeding the event in which the object was generated, an initiation event will then be registered, unless the encloser of the object is terminated.

Completion. A completion event should be registered whenever an object executes a time consuming action with a condition which is false in a concurrent state, unless the encloser is terminated. This is achieved if

- the change in the value of a variable is succeeded by the insertion of TEST NOTICEs representing the objects which execute time consuming actions with conditions involving the considered variable (we defer the discussion of how to do this), and

- the resumption of an interrupted action causes the QSG to register an event if necessary. This can be achieved by a TEST NOTICE or we may take advantage of the fact that the resumption is the last part of an event, i.e. that a concurrent state is in fact established, by executing the test for registration of an event as the first action in this concurrent state.

<u>Interruption.</u> The QSG should test for interruption whenever the agenda of an object may contain an interrupt which penetrates the ongoing time consuming action. This is achieved if

- the sending of an interrupt is succeeded by the registration of a test for the object receiving the interrupt, and
- the resumption of an interrupted action and the initiation of a new time consuming action causes the QSG to register an event if necessary (cf. the discussion under "Completion").

When the QSG register tests and events as described above, at most one notice is needed for each object on the lists EVENT LIST, TEST LIST and FUTURE EVENT LIST. This implies that

- when a test is registered for an object whose ongoing action has a time dependent condition, a possible notice on the FUTURE EVENT LIST is removed,
- when a test fails to register an event, an EVENT NOTICE is (re)inserted in the FUTURE EVENT LIST if the condition of the ongoing action of the object is a TC and
- prior to the insertion of a TEST NOTICE for an object on the TEST LIST it is checked whether the object has a notice on the TEST LIST or the EVENT LIST or not. If it has, no new notice is inserted.

The Duration of Time Consuming Actions

We will now consider how to register a test for a possible completion event. We base our discussion on the event dependent conditions, the ECs. The extension to the TCs is trivial, provided that they satisfy the restriction stated on page 19.

A test for a possible completion event should be registered when the condition of the ongoing action of an object may have become false because of an event action.

If we want to implement ECs in general, then each assignment to a variable should register tests for the objects with ongoing actions depending upon that variable. Since most variables may be used in ECs a straightforward implementation which associates a list referring to the objects in question with each such variable is not satisfactory. An improvement can be made, if the variables which are used in ECs can be declared as belonging to a "subtype" of their original type. The addition of the list and the sideeffect of registering tests when assigning to the variable will be properties of the subtype.

One of the effects of initiating a time consuming action, i. e. a WAIT, will be to insert notices, referring to the object executing the WAIT, on the lists of the relevant variables. The relevant variables may be either all the variables on which the condition depends or a subset of these decided by the QSG:

Consider the initiation of the action

```
WHILE X < Y OR I < J WAIT
```

when the relation $X < Y$ is true. It will only be necessary to insert notices on the lists of X and Y. On the other hand, the QSG may have to insert notices on the lists of I and J later, after a test executed as an effect of an assignment to X or Y (if "$X < Y$" has become false).

If several objects are WAITing on the same condition one test is executed for each object whenever the value of the condition may have become false. (Two conditions are the same if they represent identical boolean functions of the same variables. Thus objects from a class containing the sequence:

```
BEGIN INTEGER I;
              :
        WHILE I < J WAIT
              :
END
```

are not WAITing on the same condition since different "I"s are involved). It is, however, only necessary to evaluate the condition once, since no events are executed between the evaluations. To achieve this we may associate lists referring to the conditions instead of to the objects with the variables. A list of WAITing objects should then be associated with each condition. We may illustrate this in the following way

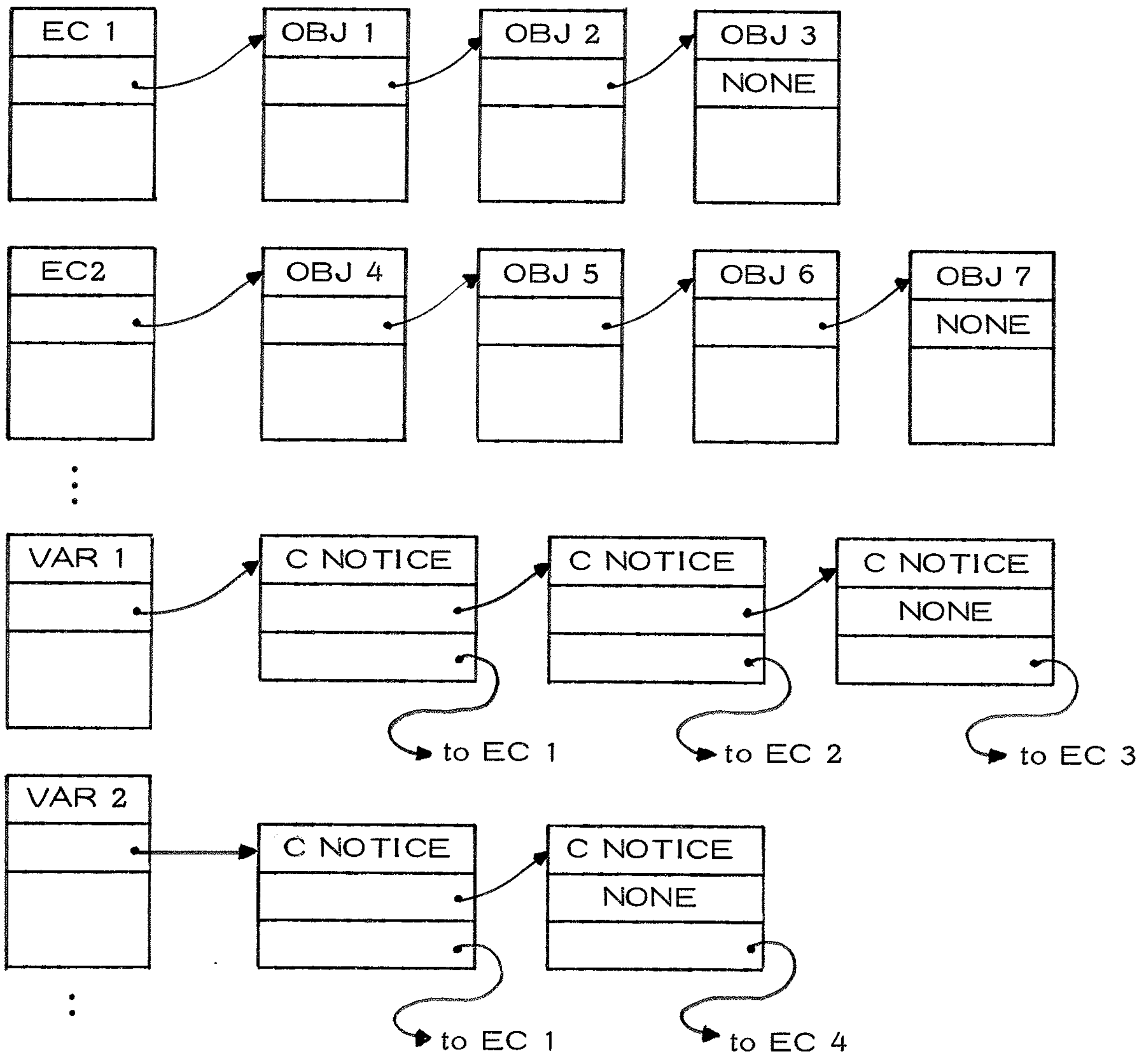

Fig. 4.1
Grouping of objects WAITing on the same condition

The objects are listed themselves, since each object is WAITing on at most one condition at a time, whereas a condition may depend on several variables and thus lists of notices have to be applied.

Different simplifications are possible for certain categories of conditions: e. g. if a condition depends upon only one variable, we may insert the condition itself on the list of the variable.

The crucial gain in efficiency is achieved by the use of conditions which do not involve waiting lists, i. e. ECs which only become false after explicit interaction with other objects (TCs may of course still schedule an event in the future).

Obviously constant conditions do not involve waiting lists. An object executing an action with a constant condition simply WAITs until it is interrupted. It then executes the sequence described by

- the postponement clause,
- the interrupt and
- the resumption clause

after which it resumes WAITing or ADVANCEs to the action following the interrupted WAIT (cf. the description of the keyword "ADVANCE" p. 8).

To describe another category of conditions, which do not involve waiting lists, we would like to be able to specify, that a variable is modifiable only by the object in which it is declared. We say that such a variable is strictly observable.

If an object executes a WAIT with a condition depending only on such variables declared in the object itself, then the value of the condition may change only if the WAITing object itself executes some actions, i. e. if it is interrupted. It is thus only necessary to check the condition (i. e. to test for completion) each time the WAITing is resumed after an interruption.

As an alternative approach to the avoidance of waiting lists we may make the registration of tests an explicit part of the program execution:

When an object changes the value of a variable which is part of a condition of an ongoing action of another object, it is the responsibility of the first object to register a test for the second object. We may illustrate this by the following example:

```
OBJECT 2 executes
WHILE I < J WAIT
```

If OBJECT 1 changes the value of I , while OBJECT 2 is executing the action described above, then OBJECT 1 should register a test for OBJECT 2, i.e.

```
OBJECT 1 executes
I:=I+DI;
OBJECT 2. REGISTER TEST
```

This scheme presupposes, that whenever an object changes a variable on which the condition of the ongoing action of another object depends, then the first object has access to the second object. In nested systems in general this is not the case. However, for the systems which we may portray by the execution of a SIMULA program, it is "structurally" possible, i.e. the properties of the SIMULA Language do not exclude it. (This is true because these systems may only consist of a system object and objects directly enclosed by the system object, cf. p. 34)

Furthermore the scheme presupposes that all the necessary calls of REGISTER TEST are made. This may be quite hard to check, and alternatively we may state that the action which we in a SIMULA program describe as

```
WHILE EC WAIT
```

(i.e. by a procedure call like WAIT WHILE (EC)) in fact portrays an action described by the pseudo-DELTA imperative

```
WHILE EC' WAIT
```

where EC' is a variable not accessible to the user, which is assigned the value EC prior to each actual evaluation of the condition by the QSG, i.e. prior to

- the initiation and resumption of the action and

- the execution of a test because of a notice on the TEST LIST (this notice was placed on the list by the sending of an interrupt or explicitly by the procedure REGISTER TEST).

5. IMPLEMENTATION IN SIMULA

Projection or expansion

In sections 3 and 4 we have derived a set of concepts by projection from a truly parallel, continuous state transformation environment into a quasiparallel, model time dependent environment.

When we implement these concepts in SIMULA we expand the SIMULA Language with its time-invariant sequencing defined in terms of coroutines (detached objects and prefixed blocks) by introducing the Quasiparallel System Generator sequencing according to model time.

A consequence of this approach of implementation by expansion is that we are not able to describe the sequencing of a nested system:

As it is known from the class SIMULATION, all objects manipulated by the use of the Sequencing Set, SQS, i.e. all process objects, have to be declared in the head of the block prefixed by SIMULATION, and this restriction holds in general (1).

<u>In order to exploit the full power of nested systems as a structuring tool we must describe the sequencing at a more fundamental level than SIMULA itself</u>.

A second, minor problem concerns the organization of the agenda. The interrupts placed on the agenda of an object may be declared in other objects and even at different levels of the system. In SIMULA a list structure like the agenda may only be built from objects by means of references. But references may not go from an outer level to an inner, i.e. not from outside an object to objects declared inside that object.

To solve these problems we need to be able to manipulate entities representing objects and interrupt procedures by the use of "entity references", i.e. references with a qualification common to all entities.

---

(1) One process object, MAIN, impersonates the prefixed block in such a way that the actions of the prefixed block are effectively scheduled according to model time.

We will not pursue the subject here. As stated above it should be treated as an integrated part of the language definition and implemented outside the scope of the language.

Implementation using SIMULATION

In this subsection we discuss the restrictions necessary when we implement the derived concepts using SIMULATION. As we shall see these are essentially the two restrictions already treated (i.e. no nesting of objects and limited possibilities of using interrupts). Nothing more is obtained by avoiding the use of SIMULATION and building directly on SIMSET or SIMULA.

We want to implement a subclass of SIMULATION in which we have

- the DELTA object structure with activity stack and agenda
- sequencing by means of time consuming actions and interrupts.

The DELTA structured objects are introduced through a common prefix, DSO, which in turn is a subclass of the class process. The DSO prefix contains a reference to a "head" object representing the agenda of the DSO. The activity stacks of the DSOs are implemented by means of the "call" mechanism. The actions described at the DSO level consist of the generation of the head of the agenda, default initialization of the priority values and the insertion of a TEST NOTICE.

The time consuming actions are built from SIMULATION's "HOLD" and "PASSIVATE". The actions are implemented in a way that incorporates the necessary registration of events when an action is initiated and when it is resumed after interruption.

We would like to describe (i.e. declare) the tasks to be used as interrupts both within the subclasses of the class DSO (which we name the subDSOs) and at the same level as these, i.e. as attributes of the block representing the system object.

However, if we try to allow the declaration of procedures within the subDSOs to describe the tasks to be used as interrupts we end up with some major restrictions:

A notice (i.e. an object) on the agenda of an object from a subDSO must distinctly indicate its associated procedure describing the interrupt represented by the notice. This requires a reference to the object, containing the declaration of the procedure, with a qualification equal or inner to the prefix level where the procedure was specified as an attribute.

But the notices are introduced at the same level as the class DSO, whereas the considered procedure is declared in a subDSO. Consequently we have to introduce such procedures as virtuals in the DSO prefix. This places a limit on the number of procedures which may be used as interrupts within each subDSO and imposes a certain amount of bookkeeping on the user, since the names of the virtual procedures are fixed regardless of their actual use.

Instead we confine ourselves to declarations as attributes of the system object. We declare a class TASK to be used as a prefix to the description of the tasks which we want to use as interrupts. In this case we are able to list the interrupts themselves on the agendas. The execution of an interrupt consists of a "call" of the corresponding object.

Our last job is to simulate the task of the Quasiparallel SG by the use of the Sequencing Set, SQS, of SIMULATION.

Examining the operation of the QSG we observe that

- it first empties the TEST LIST,
- then takes one notice from the EVENT LIST and executes the corresponding event.
- The execution of this event may produce some TEST NOTICEs in which case the QSG again empties the TEST LIST.

- When both the TEST LIST and the EVENT LIST are empty some notices are transferred from the FUTURE EVENT LIST to the EVENT LIST.

This implies that we may represent the three lists in the following way:

SQS ≡ TEST LIST <u>CONC</u> EVENT LIST <u>CONC</u> FUTURE EVENT LIST

(where <u>CONC</u> stands for concatenation).

The notices on the FUTURE EVENT LIST are characterized by having an EVENT TIME greater than the current value of TIME. We thus insert a notice on the EVENT LIST by placing it immediately after the last notice whose EVENT TIME is equal to TIME. Notices on the TEST LIST are inserted in the front of the SQS.

When TIME is increased, because TEST LIST and EVENT LIST are empty, this implicitly "transfer" the notices with EVENT TIME equal to the new value of TIME to the EVENT LIST (because the notices on the FUTURE EVENT LIST are characterized by having an EVENT TIME greater than the current value of TIME).

So far we have not touched the question of waiting lists, i.e. how to decide when to place a notice on the TEST LIST because the value of a condition of an ongoing action may have become false.

<u>INTERRUPT ENVIRONMENT</u>

We have implemented a subclass of SIMULATION, named INTERRUPT ENVIRONMENT, which uses "direct" scheduling without waiting lists by means of a procedure "REGISTER TEST" as discussed in section 4.

INTERRUPT ENVIRONMENT is now being used to write simulation programs based on DELTA descriptions. In a neurophysiological research project, where a simulation program was needed, a DELTA description was made to serve as a communication tool between the neurophysiologist and the programming specialist (Kyng and Pedersen, 1974). When the DELTA description was worked out, the use of INTERRUPT ENVIRONMENT allowed a straightforward translation into a SIMULA program.

(It should be noted that it is not possible to protect the concepts developed in a "system" class like INTERRUPT ENVIRONMENT against misuse in a prefixed block, since the protection of the interface as provided for SIMSET and SIMULATION is not part of the SIMULA Language.)

6. REFERENCES

Holbæk-Hanssen, E., Håndlykken, P. and Nygaard, K. (1975):
"System Description and the DELTA Language",
DELTA Project Report NO. 4,
Norwegian Computing Center, Oslo, Norway.

Kyng, M., and Pedersen, B. Mφller (1974):
"Description of a Model of a Single Helix Pomatia Brain
Neuron and an Associated Neurophysiological Experiment",
DELTA Project Report NO. 3,
Department of Computer Science,
University of Aarhus, Denmark.

Nygaard, K. (1973):
"On the use of an Extended SIMULA in System Description",
DELTA Project Report NO. 1,
Norwegian Computing Center, Oslo, Norway.